BDS Cプログラミング

BDS C
αC

御手洗 毅 著

工学図書株式会社

BDS Cプログラミング

御手洗 毅 著

工学図書株式会社

まえがき

本書はBDS C，α-Cユーザのためのコンパイラ解説書です．

BDS Cは，8ビットCPU用のC言語としては最も有名なもので，標準的なOSといわれるCP／M上で動作し，8080（およびZ80）CPU上で動作するオブジェクトを作成します．開発時期が早かったために多くのユーザーを獲得し，CP／Mでは最も多く利用されているC言語となっています．しかしながら，8ビットCPUに適合させるためにC言語のサブセット（機能圧縮版）となっており，標準的なC言語の解説書では若干異なる部分があり学習しにくいのも事実です．

特に最近，C言語は16ビットの世界で盛んに利用されるようになったため，8ビット用のCについてあまり触れられない事が多くなりました．しかし，8ビットCPUを用いたマイコンシステムをC言語で開発したり，CP／M上で動作するプログラムを作成する場合には，BDS Cほど便利なものはありません．

また，BDS Cの兄弟にあたる「α-C」が発売されている事も特筆すべきでしょう．このα-CはBDS Cの廉価版ともいえるのですが，実際にはBDS Cからアセンブラ，デバッガーなどの比較的利用頻度の少ないファイルを取り去っただけで，事実上BDS Cと全く同じものですから学習用，練習用には格好の素材です（なおメーカーによっては，特定のパソコン向けにBDS C，α-CをOEM販売している所もあり，より安く入手する事もできます）．

BDS C，α-Cは利用環境が整備され，バグの少ない安心して使える良質なソフトウェアですが，残念ながらあまりその利用法をまとめた書籍がありません．そこで本書では，その空白を埋める目的でユーザーの立場か

らその利用法を解説しています．内容は簡単な事からかなり高度な事まで多岐に渡っていますが，初心者からシステム開発に携わる方まで参考になる箇所は多いと思います．持に，ザイログZ80ニーモニックでアセンブラを利用できるように，Z80用CASMプリプロセッサのソースリストを掲載していますのでBDS CのユーザーでZ80の機能をフルに利用したい方，α-Cのユーザーでアセンブラが使いたい方にはその利用価値は極めて高いものとなるでしょう．

また，本書では逆アセンブルリストなどをインテル，ザイログ両ニーモニックで行ない，どちらの書式に慣れた方でも理解しやすいように考慮しました．

是非とも多くの方がC言語に触れ，その素晴らしさを知って載きたいと思います

なお，本書では次のバーショを用いています．

BDS C　　バージョン1.50a

α-C　　バージョン1.51

昭和61年3月

御手洗　毅

■CP/MはDigital Research Inc.の登録商標です．

■BDS CはBD SOFTWARE, Inc.の登録商標です．
日本総代理店は㈱ライフボートです．

■α-Cは㈱ライフボートの商標です．

目　次

第1章
BDS Cコンパイラ入門

C言語は最近注目されているシステム記述用言語で，他の高級言語と比較して移植性が高く，オブジェクトも高速でハードウェアに密着したプログラムが作成しやすいため多くの用途に用いられています．

数あるC言語のなかで，BDS Cの長所は何かと問われたら，私はその実用性をまず第一に挙げたいと思います．かなり漠然とした表現ですが，コンパイルが早く，かつ生成されるオブジェクトの実行が高速です．すなわち，いらいらせずに納得できるスピードで走行するプログラムが作成できるのです．このマンマシンインターフェースの良さはユーザーにとっては非常に重要なポイントで，BASICがさんざんな不評を買いながらも，パソコンで最も多く使われている事からもお分りいただけると思います．

この実用性を優先しているという事で，BDS Cは本格的なプログラム開発用のコンパイラであると言えるのですが，価格が安いうえ，さらに廉価版であるα-Cも市販されているため初心者が初めて手にする教育用C言語という側面も同時に持っています．そこで，本章では，まずBDS Cに的を絞ってC言語の基本について解説します．ただし，C言語のすべてではなく，重要な事項についてのみ抜粋してあります．

初心者の方は本章をざっと通読し，第2章以降で実際にBDS C（あるいはα-C）を使いながら，分からない所を読みかえしてみるという方法で学習するのが良いでしょう．

なお，既にC言語の経験のある方には特に必要のない章です．読み飛ばしてください．

1-1 C言語の基本的事項

Cは関数型の言語で，プログラムはすべて機能ごとに関数に分け，ひとつの関数からさらに別の関数を呼び出す形で大きなプログラムを作成していきます．関数というのはサブルーチンそのものですが，引数，戻り値のフォーマットが固定されているため，独立性が高く，すっきりした見やすいプログラムを書く事ができます．

また，標準的にマクロプロセッサを持っており，＃defineという擬似命令によって，文字列や数置を判りやすいシンボルに置き換えたり，マクロ展開させる事ができるので，プログラムの可読性はかなり高くなります．

標準的なプログラムのスタイルは次のようになります．

マクロ定義
外部変数の宣言

⬇

関数の定義 (1)

⬇

関数の定義 (2)

⬇

マクロ定義と同時に外部変数の宣言を行ないますが，これは各関数で共通に使う（使える）変数を宣言するものです．

(1) 変数の種類

BDS Cは基本的に次の3つの種類の変数（データ型）を持っており，使う前に必ず宣言する必要があります．

char　　文字を表現するデータ型（8ビット）
　　　　0～225

int	符号付き整数を表現するデータ型（16ビット） −32768～32767
unsigned	符号なし整数を表現するデータ型（16ビット） 0～65535

標準Cには，この他実数を示すfloat，倍精度実数を示すdouble などがありますが，BDS Cにはありません．

これらの変数は，次の2種類の記憶クラスを持っており，その時々で使い分ける必要があります．

(a) 外部変数

外部変数は大域変数，エクスターミナル(external)変数，グローバル(global)変数，パブリック（public）変数などと呼ばれるものと同じで，プログラム中すべての関数から共通に読み書きできる変数の事です．BASICなどでは変数はすべてこの外部変数です．

(b) ローカル変数

ローカル変数は局所変数，自動（auto）変数などと呼ばれ，C言語の特徴の一つです．これは関数の内部で宣言する事によってスタックから変数領域を確保し，関数が終了すると自動的に消滅する変数で，関数の独立性を保証するだけでなく，リエントラント（reentrant, 再入可能）な構造を実現するもので，リカーシブ（recursive, 再帰的）なプログラムを作成することも可能にしてくれます．

ローカル変数は他の関数から参照する事はできません．BASIC や FORTRANにはこの記憶クラスはありません．

なお，標準的なC言語はこの他にスタティック（static)変数をもっていますが，BDS Cにはありません．

この記憶クラスの考え方は特に初心者の苦しむ所なので，余り良い例では

ありませんが，CP／Mシステムを例に説明してみましょう

CP／M上で動作するプログラムから見た場合，システムコールだけを使う限り，画面上になにかを出力しようとしても，そのカーソルの位置を検出する事はできません．(エスケープシーケンスで可能な場合もあります)これはカーソル位置をしめす変数が無いのではなく，存在しているが見えないだけであり，ユーザープログラム実行中もその値は保持されています．ところが，デバイスの割付け状況を示すIOバイトは3番地と定まっていますから，簡単にその内容を参照したり，変更したりする事が可能です．即ち，カーソル位置を示す変数はスタティック変数，IOバイトは外部変数という事になります．

さてそれでは，ユーザープログラムの中で用いられる一般の変数は何でしょう．CP／Mから見る限り，その変数はユーザープログラム実行中にのみ意味があり(存在し)，実行が終了すると意味が無くなってしまいます．(存在しないのと同じです）これは，局所（ローカル）変数の特徴です．勿論，これは非常にマクロ的な視点から見た話です．

なお，記憶クラスの指定は文脈によって行ないます．宣言が関数外であれば外部変数，関数内であればローカル変数となります．従って一般には外部変数の宣言はプログラムの先頭部分にまとめるか，マクロ定義とあわせて別ファイルにしておきます．別ファイルにした場合，このファイルをヘッダーファイルと呼びファイル名の拡張子を“.h”にしておきます．

なおプログラム本体を2つに分ける時は共通のヘッダーファイルを作る必要があります．

(2) 関数の定義

関数は，次のようなフォーマットで定義します．

```
int  function(arg1,arg2)
int     arg1;
char    arg2;
{
```

```
        <プログラム本体>
    }
```

functionは関数名で，BDS Cの場合最初の8文字までが有効となります．

arg1，arg2は引数です．関数名の次の括弧の中に引数名を必要な数だけカンマで区切って列挙します．次の行でint，charと宣言しているのはデータタイプですが，これらの引数はBDS Cでは{ }内で宣言するローカル変数と全く同様に扱う事ができます．いわば，引数とは他の関数によって初期値が設定されているローカル変数と考えて良いでしょう．従って読み書きは自由にできますが，関数が終了すると自動的に消滅します．なお，変数および引数の宣言の際は行の最後に必ず；(セミコロン)を付けます．プログラムの本体部分は{ }で括り，戻り値がある時は

```
    return(a) ;
```

でaの内容を戻り値としてこの関数を終了します．関数名の前のintはこの戻り値の型を示しています．intの場合は省略が可能ですがその他の場合は示すのが原則です．

```
    char func( )
    int *func( )
```

次に関数の呼出しは次のように行ないます．

```
    x=function(y,z) ;
```

これにより，引数y，zを設定してfunctionを実行し，戻り値をxに格納します．関数の戻り値は8ビット値あるいは16ビット値1つだけしか許されません．

なお，特別な関数としてmainがあります．これは，Cプログラムが起動した時に最初に呼び出される関数で，プログラム中に必ずなければならないものです．

（3）　演算子

Cでは他の言語と異なり，代入文自身も値を持つ事ができます．これは例えば，

```
x=function(y,z);
w=x;
```

というようなプログラムがあった場合，

```
w=x=function(y,z);
```

と，1行にまとめられる事を示しています．これはプログラムのタイプ量を減らし，より高速に実行するオブジェクトを生成する効果があります．このような工夫はCでは随所にあり，特に演算子に顕著に見られます．そのため，Cで用いる演算子はBASICなどと似通ってはいるものの，異なる点もかなりあります．重要なものは覚えておく必要があるでしょう．

演算子には，算術用演算子，論理演算子，インクリメント，デクリメント演算子などがあり，これらはすべて演算用記号で表わされます．

(a)　算術演算子（＊，／，＋，－，％）

a＊bという形で用いられ，aにbを掛けた値を求めます．％はモジュロ（剰余）演算子で，a％bの場合，aをbで割った余りを求めます．

(b)　関係演算子（＞，＞＝，＜，＜＝）

a＞bなどという形で使います．この時，条件が成立していれば結果は真(1)となり，成立していなければ偽(0)となります．

(c)　等値演算子（＝＝，！＝）

＝＝は左右の項が第しいかどうかを調べます．等しければ真(1)，等しくな

なれば偽(0)になります．！＝は不一致のテストです．等しければ偽(0)，等しくなければ真(1)になります．

BASIC では一致のテストに＝を使うため，＝＝と書かねばならない場合に＝を使ってしまうケースが良くあります．初心者の方は注意してください．

(d) 論理演算子（＆＆，｜｜）

＆＆は論理積（AND）で，両方の項が両方とも0以外であれば結果を真(1)とし，どちらかでも0なら偽(0)とします．｜｜は論理和（OR）で，どちらか一方でも0以外であれば結果を真(1)とします．なお，標準Cでは＆＆は｜｜より高い優先権位を持ちますが，BDS C では同じです．

(e) インクリメント，デクリメント演算子（＋＋，－－）

これらの演算子はC言語独特のもので，＋＋はその被演算数に1を加え，－－は1減じます．両者は全く同じように，

a＋＋　　あるいは　　＋＋a

a－－　　あるいは　　－－a

という形で用います．演算子が変数の後にある場合はaをそのままの値で評価した後，aを変化させます．前にある場合はその逆で，aを先に変化させ，変化後の値で評価します．従って，

```
a=0;
if  (++a)  { 処理 }
```

では処理が実行され，

```
a=0;
if  (a++)  { 処理 }
```

では処理は行なわれません．このインクリメント，デクリメント演算子の存在理由は主に処理の高速化にあります．プログラム中で積極的にこれらの演

算子を使うことにより、高速なオブジェクトが発生します．

(f) ビットごとの論理演算子（&，｜，^，＜＜，＞＞，～）

これらはすべてビットごとの処理をする演算子で，&は論理積（AND），｜は論理和（OR），^は，排他的論理和（XOR）をとります．

＜＜，＞＞はシフト演算子で，左項で指定したビット数だけシフトします．空いたビットには0がはいります．（コンパイラによって異なります）

～は単項演算子で，整数の1の補数を求めるのに使います．

(g) 代入演算子（=，+=，-=など）

基本的な代入演算子は“=”です．これについては特に説明する必要もないでしょう．

次に，C特有のものとして，“+=”などがあります．これはa+=1と記述するとa=a+1と同様の処理を行ないます．ある変数になんらかの処理を行なって，再びその変数に値を格納する場合に有効な演算子です．

+，-，*，／，%，＜＜，＞＞，&，^，｜の演算子は，さらに=をつけて代入演算子にするこができます．これらの演算子もその目的は処理の高速化にありますが，プログラムが短くなり，見やすくする効果があります．

(h) 条件演算子（?と:）

条件演算式により，2つの式のうちどちらを実行するかを決める演算子です．簡単なプログラム分岐にも使う事ができますが，多くの場合2つの値のうちどちらかを選ぶ際に使われます．

```
a=(x>y ? x:y)
```

とすると，aにはx，yのうち大きい方が入ります．

これらの演算子にはそれぞれ優先順位があります．*，／を+，-より先

に計算するのは習慣どおりですが，他の優先順位をすべて覚えても意味がありません．それよりも不安な場合は(　)を用い、確実な順番で計算させるようにした方が良いでしょう．

(4) プリプロセッサ

プリプロセッサとは，ソースプログラムと言語処理系の中間に位置するプログラムで，ソースファイルをコンパイラが解釈しやすいように変換します．

BDS Cではこのプリプロセッサによって，大幅な言語の拡張が行なえます．これはCを使いこなす上で重要なポイントの1つで，読みやすく判りやすいプログラムを作成するのに役立ちます．

(a) ファイルの取込み

プログラム中の任意の位置に

```
#include <filename>
```

という行を記述する事により，この行をfilenameという名のファイルの内容に置き換えてしまいます．例えば，BDS Cでは一般にプログラムの先頭で，<bdscio.h>というファイルを取り込みます．bdscio.hは，標準関数を利用する場合に取り込まねばならないヘッダーファイルですから，必ず記述するものと思ってください．

なお，ファイルネームを<　>ではなく，"　"で囲む事もできますが，ファイルを捜すディスクドライブの指定が異なります．<　>ではログインディスク(A：)から捜し，"　"ではソースファイルの存在するディスクから捜します．

(b) マクロ置換

#defineを用いて名前（シンボル）を他の文字列に置き換える事ができます．

```
#define  TRUE  1
#define  FALSE 0
```

TRUE=1
FALSE=0

によって，以降TRUEを1に，FALSEを0に置き換えます．これらはソースファイルの最後まで有効で，途中で再定義する事によって内容を変更する事もできます．また，定義中に以前の定義を使う事もできます．完全に定義を抹消する場合には#undefを使います．

なお、“”で囲むと、“から”までをすべて文字列とみなします．

```
#define   TITLE   "PROGRAM 1"
printf (TITLE);
```

また，引数付きマクロも定義可能です．例えば，

```
#define max(x,y) ((x)>(y)?(x):(y))
```

と宣言すると以降max(x, y)と記述すると，x＞y ? x: yと変換されるので，関数呼出しのオーバーヘッドを避ける事ができます．

(c) 条件コンパイル

プログラム中で条件によって特定の箇所をコンパイルしたり，しないように指定する事ができます．

```
#define  DEBUG 1

#if DEBUG
     printf ("Debug mode");
#else
     printf ("Not debug mode");
#endif
```

上記のような場合，DEBUGは1ですから#if～#else の内部がコンパイルされ，#else～#endif間は無視されます．DEBUGを0に書き換えるとその

逆になります．このような条件コンパイルはデバッグ中に必要で，完成したら不要になる部分などに使われます．＃elseは省略できます．

＃if（式）のかわりに

```
#ifdef  （識別子）
#ifndef （識別子）
```

を用いる事もできます．

(d) コメント

コメントはプリプロセッサとは呼びませんが，プログラムの重要な機能の一部です．

C言語では，／＊と＊／で囲まれた部分をコメントとみなし，一切内容を無視します．また，

```
/*...........................................................
                /* comment */
...........................................................*/
```

という形で2段にネストしたコメントも有効です．これは，コメントを含むプログラムの大きな部分を無効にする際に便利です．ただし，標準Cはネスト機能を持たないため，BDS Cではコンパイル時に“－c”というオプションをつけてネスト機能をキャンセルする事ができるようになっています．

1-2 プログラムの制御構造

(1) if～else

判定にはif～else文を用います．この文法は以下のようになります．

```
if （式）
```

```
            文−1;
    else
            文−2;
```

式の値が真(0以外)であれば文−1が実行され，もし偽(0)であれば，文−2を実行します．文−2を実行する必要がなければelseそのものを省略し，

```
    if  (式)  文−1;
```

とします．

処理内容としてはただ1つの文しか書けませんが，C言語では{ }で複数の文をまとめ文法的に1つの文と同じに扱う事ができます(複文)．

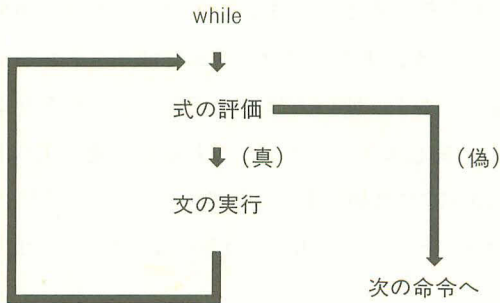
```
    if  (式)
    {
        文−1;
        文−2;
          ⋮
    }
```

複文の内部にさらにif〜else文が記述できますから，多重にネストするif〜else文も記述可能です．

(2) while

while　は繰り返しを制御します．

```
    while  (式)
        文;
```

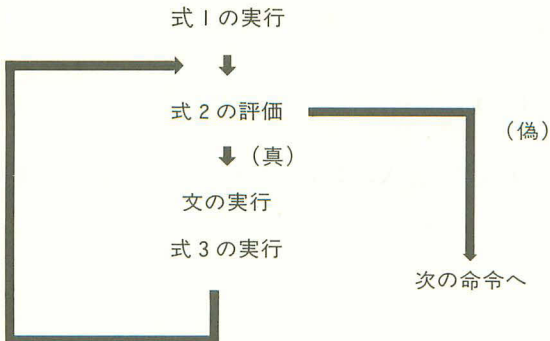
式の値が真（0以外）の時，文を繰り返し実行し，式が偽（0）の時は文を実行せずに次の行に進みます．これをフローチャートで示すと次のようになります．

複数の文を実行したい場合は{　}で囲みます．また無限ループを構成する場合はwhile(1)とします．

(3)　for

forはwhileに，初期値設定などの機能を加えたもので，

```
for　(式1;　式2;　式3)
      文;
```

という構文で使用されます．文を複文にする事もできます．

この実行の流れは，次のようなフローチャートで表わされます．

for文はwhile文で置き換える事ができるため，forを使うかwhileを使うかは好みの問題になりますが，実際のプログラムではリストが短くなる事からfor文が使われる場合が多く，使い方は知っておく必要があります．

また，forはBASICに同じステートメントがあるため使用法を間違える場合がよくあります．BASICでは終了条件としてカウンタがある値に達したかどうかしか設定できませんから，Cのfor文の方がはるかに機能は豊富です．

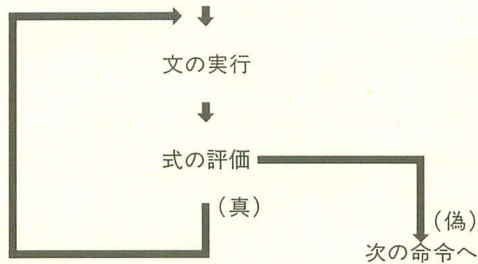
(4) do～while

whileおよびforではループ終了条件のチェックを最初に行ないますが，プログラムによっては最後に行なった方が良い時もあります．このような場合，do～whileを使うと便利です．

```
do
      文;
while (式);
```

あるいは，

```
do
{
        文-1;
        文-2;
}
while (式);
```

となります．この書式では，次のようなフローチャートになります．

(5) break

プログラムを作成していると，条件終了によらないでループから脱出したい時があります．このbreak文を使うと一番内側のループからただちに抜けます．

```
while (1)
{
    文;
    if  (条件式) break;
    文;
}
```

上記のようなプログラムでは，条件式が真になると無限ループを脱出します．ただし，breakで注意しなければならないのは一番内側のループしか抜け出ないという事で，深くネストした（入れ子になった）ループからは脱出できません．このような場合にはgoto文を使います．

(6) goto

gotoは構造化構文を破壊する機能としてPascalなどにはありませんが，gotoを利用した方が処理が簡単になる場合があるためC言語では用意されています．

```
for (…)
{
```

```
        for (…)
        {
            if (トラブル) goto error;
              :
              :
        }
    }

error:   error処理
```

上記のように，複雑な条件でのエラーや，いくつものエラー条件がある時などに，ラベルをつけて直接その場所にジャンプさせる事ができます．ラベルにはコロン（：）を付けます．

言うまでもありませんが，gotoは関数外へ飛び出させる事はできません．どうしてもそのような処理が必要な際，BDS Cではsetjmp，longjmpという関数を用いて，上位関数にじかに戻る事ができます(第3章3－1参照)．

⑺　switch

switch文は変数や式の値によって，別の処理を行なわせたい時に使う便利な構文です．

```
    switch (x)
    {
        case 0:    プログラム1;
                   break;
        case 9:    プログラム2;
                   break;
                      :
                      :
        default:   プログラム3;
    }
```

上記の例では変数xの値が0の時はプログラム1を実行し，9の時はプログラム2を実行します．そのどれでもない時はdefault：のプログラム3を実行します．プログラムは何行でも書くことができます．

switch文はこのようにcaseをどんどん並べていくだけで分岐が可能なので便利ですが，プログラムの最後にbreak文がないと，そのまま次のcaseのプログラムを実行してしまいますので注意してください．

1-3　ポインタと配列

ポインタは初心者にはなかなか理解が難しい概念の一つですが，Cがシステム記述言語として高い評価を得ている理由の一つにこのポインタがあります．メモリに密着したアセンブラ的な表現が可能で，慣れるとコンパクトで効率的なプログラムを作成する事ができるようになります．

ポインタが自由に使いこなせるようにならないとC言語をマスターしたとは言えないでしょう．

（1）　配列

配列は変数と同じようにchar，int，unsined型があり，その宣言の方法も同じです．

```
int data [10];
```

とするとint型変数10個で構成されるdataという名前の配列を確保します．int型は2バイトですから，20バイトになります．

```
配列の読み出し：　　a=data[0];
配列への書き込み：　data[1]=a;
```

配列は，10で確保するとその要素はdata［0］からdata［9］となり，data［10］は存在しません．誤って書き込んだりすると他のデータを破壊する場合もあ

りますので注意する必要があります.

2次元以上の配列は,

```
int data[4][3];
```

のように宣言しますが，BDS Cでは配列は2次元までしか使えません.

また，配列の領域は，8000H番地から確保されたとすると，1つの要素は2バイトですから，

8000H	[0] [0]	[0] [1]	[0] [2]
8006H	[1] [0]	[1] [1]	[1] [2]
800CH	[2] [0]	[2] [1]	[2] [2]
8012H	[3] [0]	[3] [1]	[3] [2]

という番地に対応します．なお，プログラム中に配列名だけを書くとそれは配列の先頭アドレスを示します.

なお，配列に限らず，変数などのアドレスは&という演算子で与えられます．例えば，&data [0][0] とすればdata [0][0] の存在するアドレスになります．これは配例“data”の先頭アドレスでもありますから,

```
&data[0][0]
```

と,

```
data
```

は等しいわけです.

また，文字列はCではchar型の配列で表わされ，最後にターミネータとして0（16進の00H）が付きます．従って“ABC”という文字列を配列にいれるためには，char data [4] ;として4文字分用意しなければなりません.

文中にじかに“ABC”と記述すると，プログラムの中に0で終端された文字列（のアスキーコード）が織り込まれ，そのアドレスが参照されます．例えば，

```
a="ABC";
```

とすると，この文を含む関数の最後に文字列そのものが設定されaには文字列“ABC”の先頭アドレスが入ります，勿論，aはアドレス変数(ポインタ)として宣言しておく必要があります．ただし，BDS Cでは文字列中にカナや漢字を使うことができません．最上位ビットを0にしてしまいますので，どうしても使いたい時には

```
str[0]=0xb1; /*ア*/
```

などのようにじかにアスキーコードを文字配列にセットするか，カナや漢字のメッセージをすべて別ファイルにしておき，プログラム中で読み込んで表示しなければなりません．また，普通のアスキー文字は‘a’，‘A’のようにクォテーションマークで囲んで記述します．

また，文字や文字列を記述する際にはコントロールコードをそのまま記述することができないので，次のような表現ができるようになっています．

LF（ラインフィード）	¥n
タブ	¥t
バックスペース	¥b
CR（キャリンリターン）	¥r
フォームフィード	¥f
¥記号（本来は＼）	¥¥
クォテーションマーク	¥'
ダブルクォテーションマーク	¥"
NULL（0）	¥0
アスキーコード	¥ddd （dddは1～3桁の8進数）

￥の後が上記以外の場合にはその￥は無視されます．なお，日本のJISコードでは￥になりますが，本来米国のアスキーコードでは，＼（バックスラッシュ）です．他のC言語のテキスト（特に翻訳もの）には＼のまま解説されているのもありますので，注意してください．

（2） ポインタ

ポインタは，他の変数のアドレスをその内容とする変数です．その宣言は例えば次のように行ないます．

```
char  *pnt;
```

これは，pntというポインタ型変数の宣言で，その内容はchar型変数を指すアドレスになります．ここで宣言したのはあくまで“pnt”というアドレス記憶用の2バイト変数であり，“＊pnt”という1バイトのchar型変数を宣言したのではありません．(16ビット以上のCPUで走行するCコンパイラでは，ポインタに3バイト以上必要とするものもあります)

式の中でpntと書くとその（内容である）アドレスを示し，＊pnt と書くとそのアドレスにより参照した内容を示します．例えば，

```
char var1,var2;     char型変数var1，var 2 を宣言
char *pnt;          ポインタ型変数pnt を宣言
var1=10;            var1に10を代入
pnt=&var1;          pntにvar1のアドレスを代入
var2=*pnt;          var2にpntの内容をアドレスとする内容を代入
```

4行めで実際に使われている変数var1のアドレスをポインタ型変数pnt に格納します．5行目では変数pntの内容をアドレスとし，その中身を変数var2に代入します．つまり，この例ではvar1の内容をそのままvar2に移す事になります．

ポインタはアセンブラでレジスタにアドレスをいれてメモリを参照する間

接アドレッシング（indirect adressing）のメカニズムをそのまま取り入れたものと考えれば良いでしょう．

〈ザイログニーモニック〉	〈インテルニーモニック〉
VAR1:　DS 1	VAR1:　DS 1　　;char var1,var2;
VAR2:　DS 1	VAR2:　DS 1
PNT:　DS 2	PNT:　DS 2　　;char *pnt;
LD A,10	MVI　A,10　　;var1=10
LD (VAR1),A	STA　VAR1
LD HL,VAR1	LXI　H,VAR1　;pnt=&var1;
LD (PNT),HL	SHLD PNT
LD HL,(PNT)	LHLD PNT　　;var2=*pnt!
LD A,(HL)	MOV　A,M
LD (VAR2),A	STA　VAR2

というオブジェクトになります．アセンブラのわかる方には参考になるでしょう（BDS Cでこのとおりコンパイルされる訳ではありません）

Cの関数ではただ1つの戻り値しか許されていませんが，どうしても2つ以上値を返したい時があります．そのような場合，外部変数を使うものも1つの方法ですが関数の独立性が損なわれます．そこで，呼び出す側で変数のアドレスを引数として与え，呼び出された側ではそれをポインタとして指す場所（要するにその変数）に結果を書きこむ方法も良く用いられます．

ポインタ変数をさらに配列にすることもあります．例えば，Cプログラムが起動された時に最初に呼び出されるmain関数では，引数にポインタ配列を使う方法で，コマンドラインの文字列を受け取る事ができます．

```
main(argc,argv)
int   argc;
char  *argv[ ];
{
```

```
        プログラム
    }
```

argc，argvという引数名は慣用的なものですから変えても構いませんが，次のような内容が格納されています．

argc　コマンドラインで与えられた単語の数・
argv　おのおのの単語へのポインタ配列の先頭アドレス

例えば，コマンドラインで次のように与えて，SAMPLEというプログラムを起動させたとすれば，

```
A>SAMPLE A BC DEF
```

main関数には次のように引数が与えられます．

argc=4
argv[1]　：文字列“A”の格納されているアドレス
argv[2]　：文字列“BC”の格納されているアドレス
argv[3]　：文字列“DEF”の格納されているアドレス

argcが4なのはSAMPLEというプログラム名そのものもコマンドラインに含まれるからで，厳密にはargv［0］が“SAMPLE”へのポインタにならなければなりません．CP／Mではユーザープログラムに制御が移った時にプログラム名を渡さないため，ユーザープログラムが自分自身のファイル名を参照する事はできません．従って，argv［0］を参照しても意味のある内容は得られません．なお，main関数への引数設定はBDS Cシステムが用意しますから，その格納アドレスが何番地になるかという事を考慮する必要はありません．

ポインタで注意しなければならないのは，ポインタ変数は宣言しただけでは値は確定しないという事です．使う前に必ず値を設定しなければなりません．

また，ポインタは変数ですが，内容がアドレスである事から他の変数とは

演算結果が異なったり，制限されている場合があります．pnt++のようなインクリメントはごく日常的に使われますが，この時，pntがchar型を指すポインタかint型を指すポインタかによって結果が異なります．char型変数が1バイトであるのに対し，int型は2バイトなので最初に指していたアドレスが0番地であったとすれば，pntのインクリメント後はchar型なら1番地，int型なら2番地となります．これは配列などを参照する時，次の要素を参照するために必要ですから，理にかなっていると言えます．これは，インクリメントだけでなく，デクリメント，及び他の加減算でも同じです．

ポインタと整数の加減算，2つのポインタの間の引算および比較は問題ありませんが，その他の演算は厳密には文法違反になります．しかしBDS Cでは違反だからと言ってできない訳ではなく，ポインタを整数に代入してしまえばすべての演算が可能です．これらの文法違反についてはC言語の厳格さとしてしばしば問題になる所で，移植性を追及する場合には十分気を付けなければなりません．しかし，専用のシステム向けのプログラムを開発する際はあまり神経質になる必要もないでしょう．

1-4　構造体

C言語は基本的にchar，int型の変数しか持ちません.(標準Cでは他の型もあります)そのため，それ以外の大きさの変数を扱いたい時には不便です．配列を使うという方法もありますが，その要素はすべて同じ変数型になってしまいます．

そこでCは構造体という概念を持っています．

```
struct list
{
    char flag;
    int  code;
} table;
```

とすると，listという名前を構造体タグとして以降3バイト(char 1バイト，

int 2バイト）のデータ型の宣言に使えるようになり，{ }の後のtableがこの変数の名前になります．listはあくまで型枠の名前です（厳密にはこのような新しい型が定義されるわけではないのですが，とりあえずこのように考えて構わないでしょう）．

上記のように型枠と構造体名を同時に宣言する事もできますが，

```
struct list
{
        char flag;
        int  code;
};

struct list table;
```

のように別々に宣言する事もできます．このほうが一般的ですが，strct listの宣言で{ }の最後にあるセミコロン(;)を忘れがちです．セミコロンは本来ここにあるべき構造体名を省略しているから必要なのだという事を知っておかないと悩む事になります．

さて，この構造体を参照する方法には2つあります．1つは直接参照で，

```
a=table.code;
```

とします．これはtableという構造体変数のcodeという要素の内容をaに代入します．tableとcodeの間にはピリオドを入れます．もう1つは間接参照で，ポインタ変数を介する方法です．

```
struct list *tblpnt;
tblpnt=&table;
a=tblpnt->code;
```

1行目でlist型構造体変数へのアドレスを内容とするポインタ変数を確保し

ます．2行めではこのポインタに構造体変数tableの先頭のアドレスを格納しています．3行目で，このポインタ変数の内容がlist型とみなし，codeに対応する要素をaに代入します．要するにcodeという要素はtableの先頭から2バイトめ（1バイトあと）にありますから，ポインタの内容に1バイト加算し，その値をアドレスとする内容を変数aに入れる事になります．この方法は構造体へのアドレスを引数として受けた関数の中で良く用いられます．

構造体は使い慣れると大変便利な機能で，データベースを作成したり，テーブル参照を行なう場合には特に有効です．構造体の配列，構造体へのポインタなど初心者には若干難しい概念もありますが，是非マスターして欲しいと思います．

以上，Cの機能をざっと解説しました．これでC言語がすべて判るとは思いませんが，最低限必要な内容ですし，良く理解して欲しいと思います．

1-5 BDS Cの特徴

BDS CはC言語のサブセット（機能圧縮版）です．そのため，C言語としての機能を100％発揮する事はできません．しかし，C言語はもともとシステム記述言語と呼ばれ，科学技術計算やOAなどの用途よりもシステムに密着したソフトウエア（OSや言語そのもの）を記述するのに向いている言語であり，これらのソフトウェアを作成する場合には，全く問題はありません．

削除されている機能は少なくありませんが，フルセット版のC言語となるとコンパイラにとっては大きな負担であり，処理系（コンパイラ）の大きさや動作速度にかなり影響を与えます．すなわち，8ビットCPU及びCP／Mに対して最適化が施されていると言えるのです．

BDS Cは1970年代の後半に最初のバージョンが発売されており，最も古いパーソナルコンピュータ用のC言語の一つです．BDS Cで作成されたアプリケーションや言語の数は多く，最も実績のあるソフトウェアの一つと言えます．この実績は安定した動作と信頼性を保証するもので，一朝一夕に得ら

られるものではありません．コンパイラの評価は機能や速度で行なわれることが多いのですが，この信頼性も大きな要素である事は言うまでもありません．

さて，具体的にBDS Cの特徴を挙げます．

(1) コンパイルが速い

コンパイラは，高級言語のソースプログラムからアセンブラのソースプログラムを作りだすものと，直接機械語のコード（オブジェクト）を作りだすものとの2種類に分けられます．前者はどのようにコンパイルされたか解析したり，部分的に書き換えたりするのには便利ですが，コンパイル→アセンブル→リンクとパスが増えるため，実行に至るまで時間がかかるという欠点があります．

BDS Cは後者のタイプに属し，しかもコンパイラが起動するとソースファイルをすべてディスクから読み出した後，オンメモリでコンパイルします．そのため処理が極めて高速で他のどのパソコン用C言語と比較しても速くコンパイルが終了します．BDS Cが他の処理系と異なり，アセンブラで書かれているという事も重要なポイントでしょう（一般にC言語はC言語そのもので作成される場合が多いのです）．

(2) オブジェクトの実行速度が速い

BDS Cは最適化によりかなり小さく，速いオブジェクトを作り出します．この詳細については第6章を参照してください．

(3) 多くの標準関数，パッケージライブラリを持っている

BDS Cは100種近い標準関数を持っており，プログラムが作りやすくなっています．

また，I/Oリダレクションを実現するDIOや，コマンドラインにワイルドマッチカードを利用可能にするWILDEXPに加え，浮動小数点，倍精度整理演算パッケージなどが標準で附属しており，様々な用途に対応させる事がで

きます．これらの内容については第3章および第4章を参照してください（α-Cには附属していないものもあります）．

(4) 専用のデバッガーがある

BDS CにはCDBという専用のデバッガーが用意されており，ソースプログラムに対応させてデバッグを行なう事ができます（α-Cにはありません）．

(5) ROM化が可能である

マイコンボードなどにROMとして実装するプログラムや，CP／M以外のOS（例えばBASIC）上で動作するプログラムを作成する事が可能です．

1-6 標準Cとの相違

C言語については，カーニハン&リッチーの，「プログラミング言語C」という書籍（参考文献1）が定本になっています．所謂標準Cというのはこの本に示されているものを言います．「プログラミング言語C」は，分りやすく書かれてはいますが，全仕様を網羅する必要から参考書としては初心者には読みにくいと思います．

BDS Cも米国での発売当初にはこの本を同梱していたという事です．

さて，BDS Cで最大の問題になるのはこの標準Cとの相違点です．C言語の移植性の高さそのものがこの本に負う所が多いため，相違点は移植性について重要な意味をもちます．CP／MはC言語の環境としては，恐らく最も小さく遅いものであり，また，BDS C自身最も古いものの1つですから，基本的に他のCコンパイラで作られたプログラムを処理できるようには作られてはいません．

しかし，その逆についてはまだ実現性があります．例えば，16ビット用のCコンパイラであるLattice CコンパイラはBDS用に作られたプログラムの移植が多少は楽になるようなコンパイルオプションを持っています．しか

しながら，これもかなり例外的で，現実には移植する事を念願においてプログラムを書く必要があるでしょう．

具体的な相違点の主なものを説明します．

(1)　変数のタイプ

BDS Cにはint，unsigned，charの3種類のタイプの変数しかありません．float（実数），double（倍精度実数），short int，long int（倍精度整数）はありません．しかし，floatとlong intは関数の形で用いる事ができます．

(2)　イニシャライザーがない

外部変数の値をコンパイル時にあらかじめ設定する事ができません．一般的なC言語では，

```
int a=0;
int array[2]={0,1};
```

という書式で，変数や配列を宣言と同時に値を代入する事ができます（外部変数とスタティック変数のみ，厳密には代入ではありません）．BDS Cではこの機能が無く，初期化のためには

文字列	……strcpy
整数（16ビット）	……initw
整数（8ビット）	……initb

の関数を使います．しかし，これは実際にはそれですむ問題ではなく，テーブルサーチなどのようなごく基本的な処理に大きなオーバーヘッドを与え，メモリ効率，処理時間，プログラムの手間いづれの点でも非常に不利になります．例えばアセンブラなどのプログラムはテーブルサーチの塊のようなものですが，BDS Cで作成する場合は工夫しないとエレガントには仕上がりません（テーブルサーチでメモリ効率を高める方法については，5-3節にアセンブラを用いた例があります）．

この制限は外部変数のアドレスが－eオプションにより任意に変えられる事からやむをえず，このようになったものです．文字配列だけは，プログラムコード中に“　”で囲む事により定数として埋め込む事ができます．

(3) スタティック変数がない

スタティック（静的）変数はプログラム内で値を保っておくために使われる記憶クラスで他の関数からは参照できないという特徴を持っています．

BDS Cでは外部変数で代替する事ができるのでまだイニシャライザ程深刻ではありませんが，プログラムモジュールの独立性が損なわれる上，変数名の管理も面倒になります．

特に，8080，Z80などのCPUではスタティック変数を用いる事により，高速化も望めるため，使えないのは大変残念だと言えるでしょう

(4) 文字変数が負の値を持たない

BDS Cでは，char型の変数は負数を持ちません．従って，

```
char a;
a=-1;
```

とすると，変数aには－1でなく255(FFH)が入ります．これは，int型の変数の内部表現を考えると，当然です(－1はFFFFH)．

(5) 0による除算

0による除算と0によるmodはゼロとなります．

(6) 関数の引数の評価

BDS Cでは関数を呼び出す時に引数の評価は最後のものから行なわれます．すなわち，

```
function (a[i],i++);
```

という場合，i＋＋が先に実行され，それからa[i]をセットします．iが

0であれば,

```
function (a[1],0);
```

と同じです．これは，標準Cでも厳密に規定されていない部分で，他のコンパイラにかける可能性がある時は，

```
function (a[i+1],i);
i++;
```

などのようにしておくべきです．

(7) キーワード

BDS Cでは，スタティック変数が無い事などから一部キーワードが異なっています．次のキーワードは変数名としては使う事ができません．

```
int       if      sizeof char
register  begin   struct else
end       union   for    void
unsigned  do      goto   while
return    switch  break  case
continue  default
```

(8) 大文字と小文字

標準のC言語では大文字と小文字を区別します．BDS Cでは変数，構造体，共用体，配列名では区別しますが，関数名では区別しません．キーワードはすべて小文字か大文字ならキーワードとみなしますが，混在している場合は認められません．

なおC言語では，慣用的に殆どの部分を小文字で記述し，定数を大文字で記述します．

(9) 標準ヘッダーファイル

標準Cではプログラム先頭で読み込むヘッダーファイル名は“stdio.h”で，多くのCコンパイラもこの名前を踏襲しています．しかし，BDS Cでは“bdscio.h”です．

コラム1

〈BDS C, α-Cの日本語対応〉

BDS C，α-Cは残念なことに日本語（カナ，漢字）に対応していません．外国製のソフトウェアにはこのようなものが大変多いのですが，実際にソフトウェアを作成する際にはこれらが大きな障害になることも少なくありません．例えば，MACRO-80のような有名なアセンブラですら，文字列は勿論，コメントにもカナや漢字を使う事ができません．

BDS C，α-Cの場合，強制的にMSBを0にしているだけなので，簡単にパッチ（つぎ）を当てることで，文字列が英数字に化けるのを防ぎ，カナ，漢字を使うことができるようになります．次の番地の内容を7FHから，FFHに変更してください．

BDS C	v1.50a	27C7H番地
α-C	v1.51	27F1H番地

ただし，この変更により入力ファイルのチェックがされなくなり，マクロや変数名・関数名にカナ，漢字を使った場合の動作が保証されなくなります．あくまで，文字，文字列，コメントにのみ日本語が使えるようになるだけだと考えてください．

第 2 章

コンパイラの使い方

BASICからC言語に入った方にとって，まず最初にとまどうのはCがコンパイラ言語であり，CP/MというOSのもとで動作させるいう点でしょう．C言語のプログラムの見方は判っても，BDS Cを動作させるにはまた別の問題があり，説明書をみなければわからない事がたくさんあります．

本章では，BDS Cを使う上で必要な事項について説明しています．CP/Mについては枚数の関係から深く触れる事はできませんが，わからない事があったらマニュアル，参考書などで研究される事をおすすめします．

2-1　コンパイルに必要なファイル

BDS Cのマスターディスクには60数個，α-Cでも30以上のファイルが含まれており，これだけでディスクは一杯になってしまい，自分が使える領域は余り残りません．そこで，まず最初に必要なファイルだけをまとめた作業用のディスクを作ります．

BDS C，α-Cともシステムとして必要なファイルは次の通りです．

表2-1　コンパイルに必要なファイル

ファイル	内容
CC. COM	コンパイラ本体(1)
CC2. COM	コンパイラ本体(2)
C. CCC	ランタイムパッケージ
DEFF. CRL	標準関数ライブラリ(1)
DEFF2. CRL	標準関数ライブラリ(2)
CLINK. COM	リンカー
BDSCIO. H	標準ヘッダーファイル

勿論，これらのファイルに加えてCP／Mのユーティリティプログラムなども必要ですから作業用ディスクはかなり一杯になってしまいます．容量の少ないドライブ1基で使う場合はかなり内容を吟味しなければならないでしょう．

実用的には最低320Kバイトのドライブが2基は必要です．本書では320バイト（2D）の5インチドライブ2基を持つシステムを用いています．今回作成したBDS Cの作業用システムディスクの内容の例を紹介しておきます．

表2-2 BDS C作業用システムディスクの内容

ファイル	説明
BDSCIO.H	
C.CCC	
CC.COM	
CC2.COM	
CLINK.COM	
DEFF.CRL	
DEFF2.CRL	
PIP.COM	ファイル転送ユーティリティ
STAT.COM	ファイル管理プログラム
SUBMIT.COM	バッチ処理プログラム
WM.COM	スクリーンエディター
XSUB.COM	バッチ処理プログラム

表2-2の中でWM.COMはスクリーンエディターのワードマスターです．ワードマスターはプログラム開発用に良く用いられますが，CP/Mで標準装備しているソフトウェアではありませんから，なければ他のエディターを用意する必要があります．CP/M標準のED.COMでも問題はありませんが，スクリーンエディターの方がはるかに効率は上がります．

なお，ファイルの内容は必要最低限のものです．BDS Cのユーティリティプログラム（CLIBやCDBなど）も後で使う事になりますが，使う方によっても用途によっても異なってきますのでとりあえず，目安だと考えてください．

作業用ディスクは以下のようにCP/Mの機能を利用して作成してください．

---

(1)ニューディスクをフォーマットします．

(2)sysgenコマンドにより，CP/Mシステムをディスクに書き込みます．

(3)pipコマンドにより表2-2のファイルを作業用ディスクにコピーします．

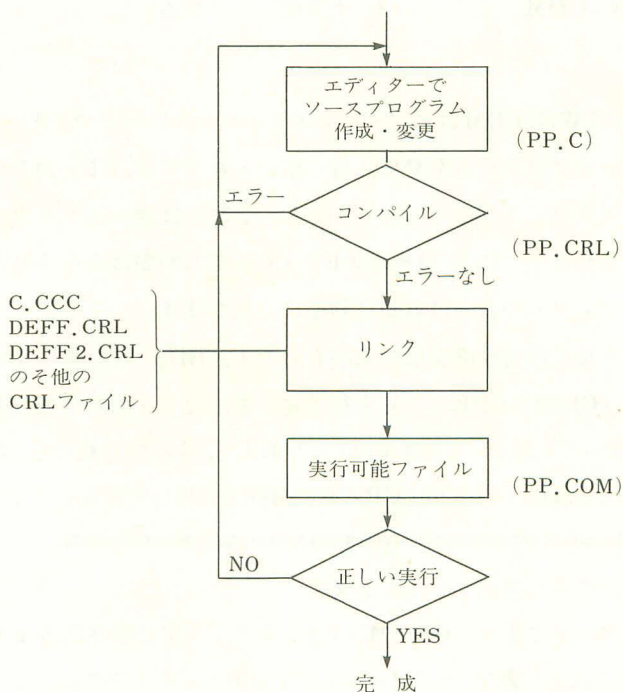
2-2　プログラムの作成

BDS C，α-Cはコンパイラ言語ですから，そのプログラム作成手順は図2-3のようになります．

まず，エディターを用いてソースプログラムを作成します．ソースプログラムとはおおもとのプログラムですが，関数単位で2個以上に分割できますから長い時には適当に分けておきます．

サンプルとして，リスト2-4を作成してみましょう．

プログラム名は，“PP.C”としておきます．

図2-3　プログラム作成の手順

リスト2-4　サンプルプログラム“PP.C”

```
#include        <dio.c>

main(argc,argv)
int     argc;
char    **argv;
{
        int     c;
        dioinit (&argc,argv);

        while  ((c = getchar ()) != EOF)
                putchar (c);

        dioflush ();
}
```

さて，PP.Cは短いプログラムですが，なかなか楽しく汎用性の高いものです．BDS Cに詳しい方以外はどういうプログラムなのか判らないと思いますので，ここで説明をしておきます．

8，9行目の

　　while ((c=getchar())!=EOF) putchar(c);

が実際に処理を行なう本体部分です．この中でcというのは6行めで指定しているようにint(整数)型の変数で，getcharという関数から得られた値を代入しています．このままこの文の値は定数EOFと比較され，EOF（End of File，ファイル終了コード，－1）でなければその値をputchar関数へ出力します．この動作を変数cの値がEOFになるまでずっと繰り返します．

さて，getchar，putcharですがこの2つはC言語の標準関数の中でも特別な入出力が規定されている関数で，標準入力，標準出力と呼ばれます．つまり，特に指定しない限り，getcharは呼ばれるたびにキーボードから入力される文字（アスキーコード）を1つずつ返し，putcharは与えられた値の文字をコンソール（ディスプイイ）上に表示します．

従って，このプログラムはキーボードから何かをタイプすると，それをそ

のままコンソールに表示するプログラムです．ところが，特別の指定をすると，その入出力の方向を切り換えて使う事ができます．言葉ではなかなか実感が沸きません．実際に作成してみましょう．

操作例2-5 PP.Cのコンパイル

〈注〉アンダラインのある所がタイプインする部分です．

```
A>pip a:=b:dio.*   ← ドライブBにBDS C（α-C）のマスターディスクを入れ，DIO.CとDIO.Hを作業ディスクに転送しておきます．

COPYING -
DIO.C
DIO.H

A>dir

A: PIP      COM : STAT     COM : SUBMIT   COM : XSUB     COM
A: CC       COM : CC2      COM : C        CCC : DEFF     CRL
A: DEFF2    CRL : CLINK    COM : BDSCIO   H   : WM       COM
A: PP       C   : DIO      C   : DIO      H
   ↑ PP.Cのファイル
A>cc pp   ← PP.Cのコンパイル

BD Software C Compiler v1.50a (part I)
  31K unused
BD Software C Compiler v1.50 (part II)
  29K to spare

A>clink pp   ← PPをリンクする．

BD Software C Linker   v1.50
Linkage complete
  40K left over

A>dir pp.*

A: PP       C   : PP       CRL : PP       COM   ← コンパイルとリンクで新しいファイルが2つ作られている．PP.COMが実行可能ファイル

A>pp   ← PP.COMを実行する．

this is test.
this is test.
My name is Tsuyoshi Mitarai.
My name is Tsuyoshi Mitarai.
   ↑ キーボードから入力してリターンすると，入力した文字がそのまま表示される．
^Z   ← ファイル終了コード(コントロールZ)
```

```
A>pp >mitarai          ← 出力をmitaraiというファイルに指定する.

This is a simple file. ┐
^Z                     ┘ テキストをキーボードから入力する.

A>type mitarai         ┐ mitaraiというファイルに先程入力した
                       │ テキストが格納されている.
This is a simple file. ┘

A>pp <pp.c             ← 入力をPP.Cに指定すると, ファイルの
                         内容をコンソールに表示する.

#include        <dio.c>

main(argc,argv)

int     argc;
char    **argv;

{
        int     c;
        dioinit (&argc,argv);

        while  ((c = getchar ()) != EOF)  putchar (c);

        dioflush ();
}

A>
```

最初にDIO.Cと, DIO.HをディスクAにPIPコマンドで転送しているのは, PP.Cの1行目に

　　＃include ＜dio.c＞

という指定があるためです. PP.Cをコンパイルすると自動的にディスクからファイル"DIO.C"を読み込み, 一緒に処理します. このDIOというファイルは実はダイレクトインプットアウトプットの略でコマンドラインで入出力

先を変更できるようにするライブラリです（詳細については第4章のDIOパッケージの項を参照してください）.

コンパイルは，ソースファイルPP.Cに対し，

```
A>CC PP
```

とします．これによりコンパイラCC.COMを起動します．このCC.COMはコンパイラの前半部で，プリプロセッサの部分です．パーザ(parser, 字句解析部）とも呼ばれる事があります．すなわち，#define，#includeなどの処理をし，コメントを削除します．この処理が終了後，CC.COMは自動的にCC2.COMを起動し，実際のオブジェクト作成作業に移ります．

コンパイル処理により，PP.CRLというファイルが作成されます．CRLファイルはBDS C専用のリロケータブル（再配置可能）なフォーマットのオブジェクトファイルで，そのまま実行させる事はできません．これを，実行形式のファイル(COMファイル)に変換するにはCLINKを使います．

```
A>CLINK PP
```

がその書式です．これでPP.COMという実行ファイルが作成されます．

リンクというのは複数のオブジェクトを合成し，実行可能な最終ファイルを作成する作業です．この場合はファイルがPP.Cしかありませんから一見無駄なようですが，すこし大きなプログラムになると開発効率を上げるために不可欠な作業になってきます．

さて，プログラムが完成したら，実行してみます．操作例2-5にもあるように，

```
A>PP >file
```

とすると，入力はキーボードのままfileとしてディスクに出力を保存する事ができます．プログラムを終了する時はコントロールZ（1AH:ファイル終了コード）を押します．また，

A>PP <file

とすると，fileの内容をディスプレイに書き出す事ができます．

このような機能を入出力リダイレクション（redirection）と呼び，UNIXやMS-DOS（v2.0以上）ではOSでサポートされています．CP／Mではこのようにプログラム中にそのような機能を持たせて，似たような機能を実現する事になりますが，こんな短いプログラムでも，typeやpipコマンドと同じような事ができるようになります．

さて，最も簡単なコンパイル例についてはお判り戴けたと思いますが，一般に，BDS Cのプログラムはリスト2-6のように書きます．これを“PP2.C”とします．

リスト2-6 PP2.C

```
#include        <bdscio.h>
#include        <dio.h>

main(argc,argv)

int     argc;
char    **argv;

{
        int     c;
        dioinit (&argc,argv);

        while  ((c = getchar ()) != EOF)  putchar (c);

        dioflush ();
}
```

最初の2行が異なっているだけで，他は全く同じです．

第1行の

```
#include <bdscio.h>
```

は，ほぼ固定です．このBDS C標準ヘッダーファイルbdscio.hは標準関数を利用する際に必要なもので，必ず入れるものと考えてください．これがリスト2-4の中になかったのはDIO.Cの中に含まれていたからです．

第2行の

```
#include <dio.h>
```

はDIOプログラムと変数を共通にアクセス可能とし，かつダブらないようにするためのもので，dio.hはDIOを利用する際に必ず用いる専用ヘッダーファイルです．PP2.Cのコンパイルとリンクは次のように行ないます．

操作例2-7　分割コンパイルの例

```
A>cc dio ←――――――― DIO.Cを先にコンパイル.

BD Software C Compiler v1.50a (part I)
  31K elbowroom
BD Software C Compiler v1.50 (part II)
  29K to spare

A>dir dio.*

A: DIO     C  : DIO     H  : DIO     CRL ←――――― DIO.CRLが作成される.

A>cc pp2 ←――――――― PP2.Cをコンパイル.

BD Software C Compiler v1.50a (part I)
  35K elbowroom
BD Software C Compiler v1.50 (part II)
  32K to spare

A>dir *.crl
```

```
A: DEFF     CRL : DEFF2    CRL : DIO      CRL : PP2      CRL
                                                   ←PP2.CRLが作成される.
A>clink pp2 dio ←―――――――――PP2とDIOをリンクする.

BD Software C Linker   v1.50
Linkage complete
  40K left over

A>dir
A: PIP      COM : STAT     COM : SUBMIT   COM : XSUB     COM
A: CC       COM : CC2      COM : C        CCC : DEFF     CRL
A: DEFF2    CRL : CLINK    COM : BDSCIO   H   : WM       COM
A: PP       C   : DIO      C   : DIO      H   : PP       CRL
A: PP       COM : PP2      C   : DIO      CRL : PP2      CRL
A: PP2      COM ←―――――――――PP2.COMが作成された.

A>pp2 <pp2.c ←――――――――――機能はPP.COMと全く同じである.
#include        <bdscio.h>
#include        <dio.h>

main(argc,argv)

int     argc;
char    **argv;

{
        int     c;
        dioinit (&argc,argv);

        while  ((c = getchar ()) != EOF)  putchar (c);

        dioflush ();
}

A>
```

今回のコンパイル例の最も大きな特徴はPP 2 . CとDIO. Cをそれぞれ別にコンパイルしている事です．従って，PP 2 . CRLおよびDIO. CRLという2つのCRLファイルが作成されるわけです．リンクの書式は，

A>CLINK PP2 DIO

となります．CLINKの次にリンクするファイル名をスペースをあけて列挙す

ると，最初にリンクするファイルと同じ名前でCOMファイルが作成されます．3つ以上ファイルがある場合も全く同じです．

このように分割コンパイルする利点はファイル1つ当たりのコンパイル時間が早くなり，デバッグが便利になる事が挙げられますが，BDS Cではソースプログラムを一度にメモリに取り込み，その後の処理をすべてオンメモリで行なうため，ソースプログラムの大きさそのものに制限があります．そのため，大きなプログラムを作る際には積極的に分割しなければならない場合もあります．

さて，これで完成したPP2.COMは機能的にはPP.COMと全く同じものです．完全に同一のオブジェクトにこそなりませんが，完成したプログラムはバイト数も同じで分割した事による機能，サイズのデメリットは全くありません．あるとすれば，ほんの少しタイピングの量が増える事でしょう．これもサブミットファイルにしておいて，自動的に行なうようにすれば，むしろタイプ量は減ります．

動作させるパソコンのCP/Mのメモリサイズにより，一度に取り込めるソースプログラムの大きさは異なりますが，ちょっとしたプログラムならばまず分割する必要はないでしょう．

2-3　コンパイルの詳細

BDS Cではコンパイル時にコマンドラインからオプションを指定する事でいくつかの機能を追加する事ができます．

このコンパイルオプションは次のような書式で与えます．

```
A<cc prog -e3000 -p
```

オプションは“－”を伴って書き，必ずその前にスペースをあけます．

(1)　－eXXXXX　　(XXXXは16進アドレス)

オプションeは頻繁に用いるオプションで，外部変数の先頭アドレスを指

定する機能を持ちます．このオプションなしでコンパイルすると，コンパイラはリンク時に決定される外部変数エリアのアドレスを知るすべがないため，リンク時に書き込まれる115H番地（2バイト）の外部変数の先頭アドレス情報を常に参照しながら外部変数をアクセスするコードを生成します．

しかし，－eオプションによって外部変数の先頭アドレスが与えられると，そのような無駄な処理を行なわず，じかに変数アドレスをオブジェクト中に置く事ができるため，*小さく，高速なオブジェクトを発生するようになります．

コンパイラでもリンカでもオプションが与えられなかったとすると，外部変数は完成したプログラムの最終番地＋1からはじまります．従って，プログラムが完成したら，リンクした時にそのプログラムの最終アドレスをみて，もう一度オプションeを指定してコンパイルし直すのが良いでしょう．プログラムが少し小さくなり，実行速度もやや速くなります．

(2) －o

－oはプログラムの実行速度を速めるオプションです．

サブルーチンコールで行なう処理を一部マクロに置き換え，オブジェクト内にそのまま展開する方法を用いますので，速度は速くなりますが，若干プログラムサイズは大きくなります．このメカニズムについては第6章に一部が述べられています．

(3) －c

コメントのネストを禁止します．

(4) －p

ソースプログラムに対し，＃defineと＃includeを処理し，コメントを削除した内容（プリプロセッサを通った後の内容）を画面に表示します．

(5)　－aD　　（Dはドライブ番号　A～P，Z）

CC2.COMの存在するユーザーエリア，ドライブナンバーを指定します．もし，Dの所にZとタイプすると中間ファイル“.CCI”をディスクにセーブし，CC2.COMを自動的にロードしません．このCCIファイルをCC2.COMで処理するとCRLファイルが得られます．極端にディスク容量が小さい時に必要になる場合がありますが，通常は全く使う事のないオプションです．

(6)　－dD　　（Dはドライブ番号　A～P）

CRLファイルをセーブするディスクドライブを指定します．指定がなければソースファイルのあるディスクに書き込みます．

(7)　－rXX　　（XXは10進数1～2桁）

シンボルテーブルのための領域をKバイト単位で指定します．あまり使う事はありませんが，“out of symbol table space”がでたら，このオプションによりコンパイルができる場合があります．本質的にはソースファイルを分割すべきでしょう．なお，デフォルトは10（Kバイト）です．

(8)　－mXXXX　　（XXXXは16進アドレス）

ROM化プログラムを作成する場合など，プログラムのスタートアドレスを指定します．（デフォルトは勿論100）ただし，－mオプションを使う際は，ランタイムパッケージ及び標準関数をリロケートしなければなりません．詳細は第6章ROM化の項を参照してください．

(9)　－x

subit処理中にエラーが起ると“$$$.SUB”ファイルを消去し，以降のバッチ処理が行なわれないようにします．submit中でコンパイルを行なう際はなるべくこのオプションを指定するべきでしょう．

⑽ －k

CDB用のシンボルファイル(. CDB")を出力し，プログラム中にトレース用のリスタート命令を挿入します．

あくまでデバッグ用ですので，それ以外の時に－kオプションを指定してはいけません．－kオプションコンパイルしたCRLファイルを通常の方法でリンクし実行すると暴走します．

なお，コンパイル時にはエラーが大量に発生することがあります．殆どの場合，それは最初にでたエラーによって連鎖的に発生したものなので，先頭にでるエラーを修正してコンパイルを繰り返してください．

2-4 リンクの詳細

CLINKは1つ以上のCRLファイルをリンク（合成）し，実行可能ファイルを作る機能を持っています．CLINKで重要なのは，リンクファイルとライブラリファイルを明確に分けている事で，指定したファイル中に使っていて定義されていない関数があると，ライブラリファイルから捜し，あれば自動的にその関数だけを抜き出し，リンクします．

このライブラリファイルとして，DEFF．CRL，DEFF 2．CRLは固定されており，この中にはBDS Cで使用可能な標準関数のすべてが含まれています．ユーザー定義用としてDEFF 3．CRLというファイル名が予約されており，もし自分用のライブラリを作りたければこの名前にしておけば自動的にスキャンされ，リンクしてくれます．

ただ，注意しなければいけないのはスキャンの順番です．DEFF．CRL，DEFF 2．CRL，DEF 3．CRLの順に行なわれますから，もしDEFF 3．CRLにDEFF．CRL内にある関数と同じ名前の関数を設定してしまうとこの関数がリンクされる事は決してありません．また，リンクはファイルの新しい関数に出会った時に，現在未リンクの関数だけをロードする方法をとり

ますので，ある関数をリンクした時，その関数の中で，さらに別の関数を呼び出していたとすると，その関数はそれ以降にある関数群（ファイル）からしか探されません．DEFF 3 .CRLに例えばprintfを用いた関数があったとすると，それまでプログラム中で全くprintfが使われていなければ，リンクができない関数としてprintfが表示される事になります．CLINKではこの時にリターンキーを押せば，もう一度ライブラリファイルを頭から検索してくれますが，面倒ですからオプションーｆを利用する方が良いでしょう．

さて，CLINKのオプションですが，次のように与えます．

```
A>CLINK file1 file2 -s -ffile3
```

CLINKの後にリンクするファイルネームを並べ，その後にリンクオプションを指定します．要領はＣＣと全く同じです．

なお，CLINKでは必ず第1番めのファイルにmain関数がなければなりません．

(1)　ーs

リンクした関数の名前とそのリンクアドレスをコンソールに表示します．

(2)　ーf　file

fileをライブラリファイルとみなし，指定したファイル中に無い関数があればこのfileから捜します．DEFF.CRL，DEFF 2．CRLファイルより先にサーチされるため，もし標準関数と同一名の関数をライブラリからリンクしたい場合は，ーｆオプションを指定しなければなりません．

なお，２つ以上のファイルをライブラリとして指定する場合にはーｆのあとにファイル名をスペースをあけて列挙します．ファイル名に“.CRL”を付ける必要はありません．

(3)　ーeＸＸＸＸ　　（ＸＸＸＸは16進アドレス）

コンパイラのーｅオプションと考え方は同じです．外部変数のスタートア

ドレスを指定します．リンクするすべてのファイルがーeでコンパイルされていれば，このオプションを指定する必要はありません．

(4)　ーo　D：file　　(Dはドライブ名)

作成したCOMファイル名とドライブを指定します．もし,ファイル名が与えられず，ドライブ名だけが指定されるとそのドライブに最初のリンクファイルと同じファイル名で作成されます．

(5)　ーn

ノーブートオプション．CCPを破壊しないので，プログラム実行終了後リブートせず，じかにCCPに戻ります．ただし,フリーメモリは2.1Kバイト小さくなります．

(6)　ーw

SID，ZSIDコンパチブルなシンボルファイル(．SYM"ファイル)を作成します．利用した関数名とそのエントリアドレスが出力されます．

(7)　ーc　D　　(Dはドライブ名)

リンク時に必要なC．CCC，DEFF.CRL，DEFF 2．CRL，(あればDEFF 3．CRL)が存在するドライブDを指定します．

(8)　ーd　　〔"args"〕

リンク終了後，すぐにプログラムを実行します．もし,"args" を与えると，それをコマンドラインに設定した状態で実行します．

(9)　ーr XXXX　　(XXXXは16進数)

リンク時の前方参照テーブルのバイト数を指定します．"Ref table overflow" エラーが発生した時に使います．デフォルトは600(H)です．

(10)　－z

プログラム実行時に外部変数をすべてクリアするのを禁止します．

(11)　－t XXXX　　（XXXXは16進アドレス）

スタックの初期値をXXXX(H)に指定します．指定しなければ，BDOSの先頭番地になります．プログラムのROM化の際などに用います．

(12)　－l XXXX　　（XXXXは16進アドレス）

生成される“．COM”ファイルの先頭アドレスをXXXX(H)番地にします．これはプログラムのオーバーレイを行なう時に利用します．

(13)　－v

オーバーレイセグメントを作成する事を宣言し，C.CCCがプログラム先頭にリンクされる事を禁止します．

(14)　－y　sname

オーバーレイセグメントを作成する際に使用します．sname.sym というシンボルファイルを読み込んで，すでにある（親プログラム中の）関数をリンクしないようにします．

第 3 章
BDS Cライブラリ

BDS Cコンパイラには豊富なライブラリが附属しており，信頼性の高いプログラムが早く，簡単に作る事ができるようになっています．これらにはBDS Cの標準関数として自動的にリンクされるものと，第2章で使ったDIOのように積極的に使わなければならないユーティリティパッケージとがあります．

標準関数はBDS Cシステムにあらかじめ用意されている関数で，DEFF．CRL，DEFF2．CRLの中にそのCRL形式のオブジェクトが含まれています．そのため，何の宣言もせずじかにプログラム中に記述する事で自動的に必要な関数だけがリンクされます．

そのソースファイルは，STDLIB1．C，STDLIB2．C（Cのソース）と，DEFFA．CSM，DEFFB．CSM，DEFFC．CSM，DEFFD．CSM（アセンブラのソース，α-Cには附属していません）で，すべてが公開されており，気に入らない部分を修正したり，全面的に変更する事も可能です．これらの標準関数が正しく動作しない（ように思える）場合，マニュアルよりも，ソースファイルを見た方が早い時もあり，ぜひとも，これらのファイルをリストアウトし，資料化しておく事をお勧めします．

以下，標準関数について，解説します．

3-1 一般関数

一般関数は主にCP/Mとのインタフェースなどのために用意されているもので，BDS Cだけが持つ関数と考えて良いでしょう．多くの関数について他のC言語と移植性はありません．

```
exit ( )
```

オープンされているファイルをクローズし，CP/Mをリブートします．

```
int bdos (c,de)
```

CをCレジスタに，deをDEレジスタに格納して，BDOSコールを行います．BDOSからリターンした時のHLレジスタの値が戻り値になります．

```
char bios (n,c)
```

cをBCレジスタにセットしてn番めのBIOSコールを行ないます．リターン値はAレジスタの値です．

```
unsigned biosh (n,bc,de)
```

bcをBCレジスタに，deをDEレジスタにセットしてn番めのBIOSコールを行います．リターン値はHLレジスタの値です．

```
char peek(adr)
```

アドレスadrの内容を返します．

```
poke (adr,b)
```

アドレスadrにbの値の下位８ビットを格納します

```
char inp (n)
```

I／Oアドレスnから入力した値を返します．

```
outp (n,b)
```

IOアドレスn（下位８bitで評価）にbの下位８ビットを出力します．

```
pause( )
```

何かコンソール入力があるまで待ちます．

```
sleep (n)
```

CPUのクロックが4MHzの時，n／20秒待ちます．コントロールCによりリブートします．

```
int call (adr,a,h,b,d)
```

aの下位８ビットをAレジスタに，hをHLに，bをBCに，dをDEにセットしてadr番地をコールします．リターン値はHLレジスタの値です．

```
char calla (adr,a,h,b,d)
```

callと殆んど同じですが，リターン値はAレジスタの値になります．

```
int abs (n)
```

nの絶対値を返します．

```
int max (a,b)
```

2つの整数のうち，大きい方を返します．

```
srand (n)
```

nが0以外の時，rand関数を初期化します．

nが0の時，“Wait a few seconds, and type a CR:”というメッセージを表示してキー待ちとなり，キーを押すと，そのタイミングに従ってrand関数に初期値を与えます．

```
srandl (str)
```

メッセージを指定して，srand(0)と同様の処理を行ないます．

```
int rand ( )
```

1から32767までの乱数を与えます．

```
setmem (adr,count,byte)
```

アドレスadrからcountバイトだけbyteデータで満たします.

```
movmem(source,dest,count)
```

アドレスsourceからcountバイトのブロックをdestアドレスの位置に移動します．プログラム内部で，8080とZ80CPUの自動判別を行ない，Z80CPUであれば，ブロックムーブ命令を利用します.

```
qsort(base,nel,width,compar)
```

base番地から，widthバイトのnel個のデータをシェルソートします．シェルソートというのは，最初のデータとデータ数の1／2番めのデータから順に比較交換していく方法で，最終的に隣同士のデータをチェックするまで1／8，1／16と比較交換する対象データの距離を変えていきます（実際にはデータ数／(2のn乗)の整数部分をチェック距離とします).

この方法はデータ数が少なく，比較する値そのものが小さい小型のソートには比較的効率の良い方法で，大きなバッファメモリも不要です．関数名がqsortですからクイックソートアルゴリズムを用いていると思いがちですが，そうではありません．一般にクイックソートの方が効率が良いと言われますが，データの個数が少ない時には殆ど影響ありません.

さて，qsortを使う時に重要なのは，第4引数です．この引数はソート時に大小比較をするために用いる比較関数のエントリーアドレスです．この関数は自分で作成する必要があります．何故，このように面倒な事をしなければならないかと言えば，この比較関数を作成する自由が与えられる事で，文字列，整数など種類の異なるものを同一の関数でソートできるからです.

この比較関数は原則的に次のように作成します.

```
compar (x, y)
    *x>*yの時  出力1
    *x<*yの時  出力-1
    *x==*yの時出力0
```

ここで,引数x, yをcharと宣言すれば,文字(例)のソート用になり,intと宣言すれば,整数のソートとなります.

具体的に,compar関数の例を示します.

(1)
```
strcmp1(x,y)
char  *x,y;
{
      if (*x==*y) return(0);
      return ((*x>*y)?1:-1);
}
```

(2)
```
numcmp(x,y)
int   *x,*y;
{
     if (*x==*y) return(0);
     return((*x>*y)?1:-1);
}
```

(3)
```
strcmp2(x,y)
char  **x,**y;
{
      if (**x==**y) return(0);
      return((**x<**y)?1:-1);
}
```

文字配列をじかにソートする場合には(1)のような関数を使います.この関

数では頭を1文字しかチェックしませんから，abcとacdは一致とみなされ，どちらが先になるかはその時の状況で異なります．これが問題になる場合は標準関数のstrcmpをそのまま用いるのが簡単ですが，いずれにしろ数字のほうがアルファベットより手前にソートされますし，大文字と小文字も異なるものとして扱いますから，書き直す必要があります．

(2)は整数値を比較する場合です．

(3)はポインタ配列を介して，文字列をソートする場合です．つまり．文字列の頭のアドレスをしまっておくポインタ配列に対し，それが指している文字列の内容を比較する事によって，ポインタ配列の方の内容を移動します．間接的なソートを行なう事になりますが，長い文字列をソートする際には内容そのものを移動すると時間がかかりますので，処理が早くなります．また，(1)の場合は，固定長の配列しかソートできませんが，(3)のようにポインタを動かすようにしておけば，不定長の文字列に対してもソートが可能になり，非常に有効な手段となります．

このように，compar関数はその目的によって，非常に多くのバリエーションが考えられます(だからこそ，自前で作る必要があるのです)．

qsortは文字列を扱ったり，数値の分布などを扱う際にはなかなか便利な関数ですし，1行qsortと書き加えれば良い場合も多く，使い方もやさしいのですが意外と使われていないように思います．4-2節のwildexpパッケージの項のリスト4-2を参照してください．

```
int exec (prog)
chas *prog;
```

prog．COMをロードして実行します．ファイルネームに“．COM”は不要です．

```
int execl(prog,arg1,arg2,…,0)
char *prog,*arg1,*arg2,…;
```

prog. COMをロードして実行します．フィイルネームに“. COM”は不要です．

```
execv (prog,argv)
char *prog,*argv[ ];
```

80文字以上のコマンド引数を持って，prog. COMを実行します．argvは文字列のポインタ配列の先頭アドレスで，配列の最後は0とします．

```
int swapin (file,adr)
char *file;
```

fileをアドレスadrからロードします．

```
char *codend( )
```

プログラムの最終アドレス＋1の値を返します．この値は119H番地に格納されています．

```
char *externs ( )
```

外部変数エリアの先頭アドレスを返します．外部変数エリアのアドレスを指定していない場合はcodend( )で得られる値と同じになります．この値は115H番地に格納されています．

```
char *endext ( )
```

外部変数エリアの最終アドレスを返します．この値は11ＢＨ番地に格納されています．

```
char *topofmem ( )
```

プログラム走行時のCP／Mシステムで利用可能なメモリエリアの最上位アドレスを返します．一般にBDOSの先頭アドレス－1を与えますが，リンク時に－nオプションが与えられていると，それより2100バイト少なくなります．

```
char *alloc (n)
```

allocはメモリのフリーエリアからメモリを切り出す関数です．alloc（n）により，nバイトのメモリ領域へのアドレスを返します．エラー時には0を返します．

この関数は元々カーニハン＆リッチーの本（参考文献1）に出てくるもので，これをBDS C用に書き直したものです．従って，他のC処理系に移植しやすい関数の1つです．使い方は簡単ですが内部の処理はかなり複雑です．

まず，BDS Cのメモリマップを大雑把に図示すると，図3-1のようになります．

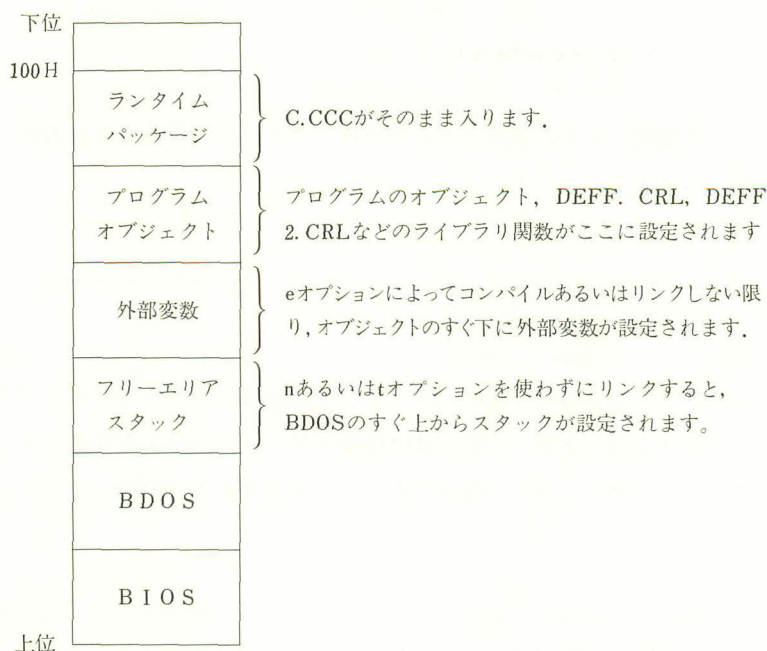



図3-1 オブジェクトのメモリ配置

CP／MのTPAは100Hから始まりますから，一般的にはプログラムは100Hから格納され，その後に外部変数が配置されます．また，上位アドレスからはプログラムで利用するスタックが伸びて来ますが，外部変数の直後からスタックで利用しない所までは空いたメモリスペースとなるわけです．このメモリエリアを管理するプログラムがallocです．

アセンブラでプログラムを作成した場合など，スタックは通常数百バイトもあれば十分ですが，BDS Cではローカル変数としてスタックを利用するためスタックはかなり深くなると考えなければなりません．プログラムの内容に依存しますが，大きな配列をローカル変数として宣言すると一時的に数10kバイトにもスタックが伸びてくる場合があります．ローカル変数の確保時にはスタックがプログラムや外部変数領域を侵すかどうかはチェックされませんから，特に大きなデータを扱う際は処理方法を考えなければなりません．

allocでは，フリーエリアを下位から上位に領域を確保していきます．この時，関数sbrkによってスタックとの距離を調べ，万一ダブってしまいそうな場合にはエラーとして処理します．スタックとぶつかるかどうかの判定は一概には決められないので，単純に1000バイトの距離が空いているかどうかでチェックします．この1000バイトという距離はプログラムの種類によって異なりますから，この距離を変える関数rsvstkも用意されています．

```
rsvstk (2000);
```

により，この距離を2000バイトに設定します．rsvstkはスタックの下限を定めると，説明される場合もありますが，これは誤りです．

allocはsbrkがフリーエリアから受り取ってきたメモリを管理しますが，呼ばれるたびにメモリにポインタを付加し，リスト構造にしておきます．このようにしておくと,不要になったメモリを関数freeによって解放し,次にallocを呼び出した時に，不要になったメモリ領域から必要なメモリが取り出せれば，その領域に割り当てる事ができます．要するにメモリの再利用をするわけです。

このallocはスピードの遅い８ビットCPUではかなりなオーバーヘッドとなりますが，移植性が高く，可変長の文字配列などを扱う際には便利なので，良く用いられます．

なお，allocを使う際には，必ず“bdscio. h”をファイル先頭でインクルードし，allocを呼び出す前に

```
_alloccp=0;
```

と外部変数_alloccpを初期化しなければなりません．BDS Cではリンク時に－zオプションを指定しない限り，外部変数はすべて0に初期化されるため無くても問題ありませんが，明確にしておいた方が良いでしょう．

allocのリスト構造については，参考文献の「1.」を参照してください．

```
free (allocptr)
```

allocによって得たメモリを解放します．allocptrはallocの呼び出し時に得たメモリエリアの先頭アドレスで，関係のないアドレスをfreeに与えると暴走します．

```
char *sbrk (n)
```

フリーエリアからnバイトのメモリを切り出す関数でallocの中で使われています．切り出したメモリの先頭アドレスを与えます．メモリが足りない時は－1を返します．

```
rsvstk(n)
```

allocの項を参照してください．

```
int setjmp (buffer)
char buffer [JBUFSIZE];
```

```
longjmp (buffer)
char buffer [JBUFSIZE];
```

longjmpはサブルーチンのネストを気にせず，一気に上位のプログラムにリターンする関数です．JBUFSIZE（6バイト，bdscio．hの中で定義されています）の大きさの文字配列を外部変数として宣言しておき，そのbufferのアドレスを引数とします．longjmpを実行する前に必ずsetjmpを実行しておく必要があります．具体的な使用法を操作例3－2に示します．

操作例 3-2

```
A>type rev.c

/****   file reverse program    ****/

#include        <bdscio.h>
#include        <dio.h>

#define RECMAX  1000    /* recursive call max count */

char    buffer[JBUFSIZE];
int     reccnt;         /* recursive call counter   */

main( argc,argv )
int     argc;
char    **argv;
{
        dioinit ( &argc, argv );
        if ( setjmp ( buffer ) != 0 )◀──────関数revでリカージョン(再帰)
        {                                    回数が多過ぎると処理を打ち
                                             切ってここに戻る.
                puts ("Too long file.");
                dioflush ();
                exit ();
        }
        reccnt = 0;◀──────────────────リカーシブコールのカウンタを
        rev ();                              リセットする.
        dioflush ();
}

rev()
{
        int     c;
        if (reccnt++ > RECMAX)          }
                        longjmp ( buffer );}◀──リカージョンの回数をカウントし,
                                                RECMAXより多くなると
                                                longjmpで強制的に打ち切る.
        if ((c = getchar ()) != EOF)
        {
                rev();
                outchar (c);
        }
}
```

```
A)cc rev
BD Software C Compiler v1.50a (part I)
  34K elbowroom
BD Software C Compiler v1.50 (part II)
  31K to spare

A)clink rev dio
BD Software C Linker   v1.50
Linkage complete
  39K left over

A)rev
abcdef0123456^Z  ←――― 手でタイプした.
6543210fedcba    ←――― 画面に逆順で表示される.

A)rev <rev.c     ←――― rev.cのソースファイルを
                        逆順に表示してみる.
}


}
;)c( rahctup
;)(ver
{
)FOE =! ))( rahcteg = c(( fi

;) reffub ( pmjgnol
)XAMCER > ++tnccer( fi
;c      tni
{
)(ver
        :
        :
        :
/* tnuoc xam llac evisrucer */  0001    XAMCER  enifed#

>h.oid< edulcni#
>h.oicsdb<      edulcni#

/****   margorp esrever elif    ****/
```

rev.cのファイル中の文字がすべて逆順に表示されている.

```
A)rev <dio.c  ←――― 長いファイルでテストしてみると,
Too long file.       setjmp,longjmp関数が正しく働いて
                     処理が打ち切られる.
```

このプログラムrve．cは再帰（リカージョン）を利用し，ファイルの内容を逆順に表示したり，ファイルに落としたりするもので，実用性はありませんが，サンプルとしては非常に楽しいものの1つです．revが再帰関数で，先頭で再帰の回数をカウントし，指定回数（1000回）以上になると強制的に再帰を打ち切り，longjmp関数で上位のmainに戻ります．main側では見掛け上setjmp関数からリターンしてきた状態になります．

setjmpはlongjmpを実行する前に必ず実行しておかねばなりませんが，この時の戻り値は0となり，longjmpからダイレクトに戻った時は0以外になります．

setjmp，longjmpはプログラム終了時に必ず実行しなければならないルーチン（例えばdioflush(　))がある時や，ネストの深いルーチンでエラーを検出した場合，再帰処理などでサブルーチンのネストそのものが管理できない場合などに用いられますが，引数を渡す事はできないため，1つの setjmpに対し，複数のlongjmpがある時，どのlongjmpから戻ってきたのかわかりません．

なお，setjmp，longjmpはあくまで上位関数に戻る時のみ利用できます．main関数以外は上位にも下位にもなりうるわけですから，万が一にも下位ルーチンに戻らないように注意してください．暴走します．

bufferは複数個作成する事で，setjmp－longjmpの対をいくつも利用する事ができます．

```
int memcmp(source,dest,length)
char *source,*dest;
unsigned length;
```

source，dest番地からのlength長のメモリをチェックし，一致した場合は1，一致しない場合は0を返す関数です．

3-2 文字入出力関数

文字入出力関数は文字，あるいは文字列の入出力を行なう関数です．基本的に標準Cの仕様に合せてありますが，異なるものもあります．

```
int getchar ( )
```

標準入力（キーボード）から1文字入力し，画面上にその文字をエコーバックします．ただし，CR（キャリッジリターン）の場合は画面にCRとLF(ラインフィード)をエコーし，入力はLFだけとなります．

なお，CP／Mのファイル終了コード，コントロールZ(1AH)は－1に変換されるため，getcharの戻り値をchar型変数に代入すると255となり，本来255のコードと区別できなくなります．

```
char ungetch (c)
```

文字cを次のgetcharの戻り値になるようにします．ただし，2回以上繰り返して行なおうとするとその文字そのものが返ります．通常時の戻り値は0です．

```
int kbhit ( )
```

キーボードが押されているかどうかを調べる関数です．押されていれば真(1)，押されていなければ，偽(0)を返します．ungetchされている場合は常に真になります．

```
putchar (c)
```

文字cを標準出力(コンソール)に出力します．LF（¥n）はCR－LFに変換します．コンソール出力時にキーボードをチェックしますので，＾S（画面出力一時停止)，＾C(リブート）が使用できます．

```
putch (c)
```

putcharと同じです．ただし，＾S，＾Cは無効です．

```
puts (str)
char *str;
```

文字列strを画面に出力します．

```
int getline(strbuf,maxlen)
char *strbuf;
int maxlen;
```

キーボードから長さmaxlenまでの１行入力を行ない，入力された文字列をstrbufに格納します．リターン値は文字数になり，CR，LFコードは格納されません．

maxlenは文字列終端用のnullバイトを含むため，実際にstrbufに格納されるのは最大maxlen－１となります．

```
char *gets (strbuf)
```

最大入力文字数が135に固定されたgetlineです．ただし，リターン値はstrbufへのポインタとなります．

```
printf(fromat,arg1,arg2,…)
char *format;
```

文字列，数値などを混合して画面に出力できる汎用の関数です．format中に含まれる変換文字に従って，arg1，arg2，…などの変数値などを出力します．

変換文字には次のようなものがあります．

d	符号付き10進数に変換します．
u	符号なし10進数に変換します．
c	下位8ビットのアスキー文字を表示します．
s	文字列を出力します．文字列は終端の0，あるいは指定された文字数までを出力します．
o	8進数で表示します．（先頭に0を付けません）
x	16進数で表示します．（先頭に0xを付けません）

これらの変換文字は，次のように使います．

```
printf ("i=%d¥n",i);
```

%が変換文字の指定で，“ ”の中にこの文字があると，その次の文字を変換文字と解釈し引数を参照します．“i=”の部分はそのまま表示されますので，もしiが5であれば，画面上に“i=5”と表示し改行します．“¥n”はLF（ラインフィード）コードで，文字列の中にあるとCRとLFの2つに変換されるため，改行になるわけです．

変換文字は，厳密には，

%[−] [0] [w] [.n] 〈変換文字〉

というフォーマットで記述され，［ ］の中は省略する事ができます．%と変換文字の間にはいるのは，

w　　文字列の長さ
n　　最大文字数
0　　余りの部分を0で埋める
　　省略されていると埋めない
－　　文字を左側に詰める
　　省略されていると，文字を右に詰める

“ａｂｃｄｅｆ”を出力すると次のようになります.

```
%4s:     "abcdef"
%8s:     "  abcdef"
%-8s:    "abcdef  "
%8.4s:   "    abcd"
%-8.4s:  "abcd    "
```

```
_spr(format,putcf,argl)
```

_sprはマニュアルには出てこない隠れた関数ですが，他の標準関数と同じようにDEFF. CRLファイル中にオブジェクトが入っており，CLINKやL2でリンクできます.

_sprはprintf，sprintf, fprintf，lprintfの中で呼び出される関数で，次の書式で用いられます.

```
_spr (fromat,putcf,argl)
```

_sprは実はprintfなどの本体部分で，%s，%cなど文字列中の制御文字すべてに対し，指定の処理をする関数です. putcfは1文字を出力する関数のアドレスで，printfの場合はputcharが指定されています. arg1はputcf関数への引数で，ファイルの出力などの場合，出力バッファ(iobuf)となります.

_sprの便利な点は新しいprintfコンパチブル関数を増設できる事で，特に次のような場合に便利です.

(a) シリアルポート，ユーザー増設のハードウェアなどへの出力関数を作成する場合

(b) DIOパッケージなどを利用する時，出力先を変えたくない記述がある場合

(c) その他デバッグ時など

printfコンパチブル関数を作成したいという場合は良くあるもので，_sprを知らなければ，sprintfを使って一旦文字列をバッファに出力してから処理する事になりますからプログラムの負担が大きくなります．printfは"stdlib2.c"の中で次のように定義されています．

```
printf (format)
char *format;
{
	int putchar ( );
	_spr (&format,putchar);
}
```

printf関数は元々引数の数が不定ですから関数側では引数の個数を知る事ができません．そこで，BDS Cでは一旦関数で受け，最初の引数(文字列)のアドレスをさらに引数として_sprを呼び出すようにしています．このようにすると，_spr側ではprintfへの複数の引数を配列の形で受け取る事ができ，処理が簡単になります．この考え方は引数の受け渡し方法を良く理解していなければ判らないので，5-2節を参照しながら考えてみてください．

さて，実際に_sprを使った例を示します．

リスト3-3　_spr関数の利用例

```
/* printf (always output to console) */

eprintf(format)
char    *format;
{
        int     putc();
        _spr (&format, &putc, 4);
}


/* PUN: printf */

#define PUNOUT  4

pprintf(format)
char    *format;
{
        int     xbdos();
        _spr (&format, &xbdos, PUNOUT);
}

xbdos(de,c)
{
        bdos(c,de);
}
```

リスト中で定義されているのは，eprintfとpprintfという関数ですが，eprintfはDIOパッケージを使っている際にも入出力の方向が変わらず，必ず画面に文字を出力します．エラーなどの表示に便利です．

pprintfはCP／Mで指定されているPUNデバイス（パンチャー）に文字を出力します．RS232Cポートなどに設定されていれば，通信プログラムなどを作る際に役立ちます．なお，pprintfで注意する事が一点あります．それはbdos関数がそのまま使えないという事です．__sprの出力先関数の書式は

```
putcf(c,arg1)
```

に固定されています．cは文字，arg1は__sprで指定した第3引数です．bdos関数はbdos（c，de）で，第1引数がファンクション番号，第2引数が

データなので順序が逆になります．そこでリストのように引数逆転用の関数xbdosを作成して正しく動作するようにしたわけです．

なお，__sprの第2引数は関数アドレスです．printfの定義例のようにputchar(　)が関数である事を明記しておくと関数名はその関数へのアドレスとして評価されます．

また，scanf, sscanfなどのため，__scnという関数もあります．scanfはprintf程使われないので省略しますが，考え方はほぼ同じです．

```
lprintf(format,arg1,arg2,…)
char *format;
```

出力がLST（プリンター）になるprintf関数です．

```
int scanf(format,arg1,arg2,…)
char *format;
```

format中の変換文字に従って，入力される文字を変換し，引数arg1, arg2, …の中に格納します．変換文字は基本的にprintfと同じですが，%uは使えません．引数は変数へのポインタでなければなりません．なお，scanfはprintfに対応する入力関数ですが，詳細は参考文献「1.」を見てください．

3-3　文字列処理関数

文字列処理関数は0で終端された文字列を処理する関数です．

```
int isalpha(c)
```

文字cがアルファベットなら真(1)，そうでなければ偽(0)を返します．

```
int isupper (c)
```

文字 c が大文字なら真，大文字以外なら偽を返します．

```
int islower (c)
```

文字 c が小文字なら，真，小文字以外なら偽を返します．

```
char isdigit (c)
```

文字 c が数字なら真，数字以外なら偽を返します．

```
int toupper (c)
```

文字 c が小文字なら大文字にして返します．それ以外ならそのまま返します．

```
char tolower (c)
```

文字 c が大文字なら小文字にして返します．それ以外ならそのまま返します．

```
int isspace (c)
```

文字 c が空白文字（スペース，タブ，LF）なら真を返し，それ以外なら，偽を返します．

```
sprintf(buffer,format,arg1,arg2,…)
char *buffer, *format;
```

printfと同様の処理をしますが,画面ではなくメモリが出力対象となり，格納する先頭アドレスはbufferとなります.

```
int sscanf (buffer,format,arg1,arg2,…)
char *buffer,*format:
```

scanfと同じ処理をしますが，入力はコンソールからではなく，bufferからになります.

```
strcat (s1,s2)
char *s1,*s2;
```

文字列s 1の後にs 2を付け加えます.s1には十分余裕がある事が必要です.

```
int strcmp (s1,s2)
char *s1, *s2;
```

文字列s 1とs 2を比較し，全く同一なら0を返します．一致しない時は文字を先頭から順に調べ，最初に異なる文字についてs 1＞s 2なら正の値を返し，s 1＜s 2なら負の値を返します.

```
strcpy (s1,s2)
char *s1,*s2;
```

文字列s 2をs 1にそのままコピーします.

```
int strlen (string)
char *string;
```

文字列stringの文字数を返します．ただし，文字列終端の0は含みません．

```
int atoi (string)
char *string;
```

stringに示されるアスキー数字を数値に変換します．もし，文字列中に数字以外のものがあると0を返します．

```
initw (array,string)
int *array;
char *string;
```

整数配列を初期化します．

```
int var [5];
initw (var,"-1,10,29,1,101");
```

のように使います．ただし，この関数は－32760を特別のフラグとして使っているため，初期値として設定する事ができません．

```
initb (array,string)
char  *array,*string;
```

文字配列を数値（0～255）で初期化します．使用方法はinitwと同じです．

```
char *index (str,substr)
char *str,*substr;
```

文字列str中からsubstrの部分を発見したらその先頭アドレスを返します．見付からなければ，－1を返します．

3-4　低レベルファイルI／O関数

低レベルファイルI／OはCP／Mの環境に影響を受け，ブロック(128バイト）単位での入出力を行ないます．ここで示す関数はfd（ファイル識別子）によって，自動的に目的のファイルに対し，処理を行ないます．このファイル識別子はopen関数実行時にシステムで決定され，戻り値として与えられます．一般にエラー時には－1を返します．

```
int open (filename,mode)
char *filename;
```

ファイルをオープンします．modeは

0　入力
1　出力
2　入出力

を指定します．関数の戻り値はファイル識別子で，read，write，seek tell，fabort，close関数でファイル名のかわりに使います．ファイルがオープンできない時は－1を返します．

```
int creat (filename)
```

同一のファイル名が既にディスク上にあれば，そのファイルを消去してから入出力モードでオープンします．戻り値はファイル識別子です．

```
int close (fd)
```

ファイル識別子fdで示されるファイルをクローズします．エラー時には－1を返します．ただし，main関数終了時，あるいはexit関数による終了時にはオープンされているすべてのファイルがクローズされます．

```
int read (fd,buf,nbl)
char *buf;
```

ファイル識別子fdで示されるファイルからbufへその内容を読み込みます．nblはブロック数で，1ブロック128バイト単位で指定します．

通常は前にread，あるいはwriteされた次の内容を読みますが，ランダムにファイルをアクセスするためにはseek関数を使います．

なお，戻り値は実際に読み込んだブロック数で，通常は引数nblと同じですが，ファイル終了時にはそれより小さくなり，すでにファイルが終了していれば0を返します．リードエラー時には－1を返します．

```
int write(fd,buf,nbl)
char *buf;
```

bufの内容をファイルに書き込みます．nblは書き込むブロック数で，戻り値は実際に書き込んだブロック数になります．

```
int seek (fd,offset,code)
```

現在オープンされているファイルのリードライト位置を示すポインタを変更します．ファイルをランダムアクセスする場合に使います．

code
0： ポインタ←offset
1： ポインタ←ポインタ＋offset
2： ポインタ←ファイルエンド＋offset
（offsetは0以下）

```
int tell (fd)
```

ファイルR／Wポインタ値を返します．

```
int unlink (filename)
```

filenameで示されるファイルを消去します．

```
int rename (filename1,filename2)
```

filename1というファイルの名前をfilename2という名前に変更します．

```
int fabort (fd)
```

ファイルをクローズせずにreadを中断します．

```
unsigned cfsize (fd)
```

ファイルのサイズをブロック数で返します．

```
int errno ( )
```

ファイルI/Oでエラーが起った場合，そのエラーコードを返します．

```
char *errmsg (errnum)
```

エラーコードに対応するエラーメッセージ（文字列）を返します．errnumは0～14です．

```
int setfcb (fcbadr, filename)
```

fcbadrで示されるFCBアドレスにfilenameをセットします．

```
char *fcbaddr (fd)
```

ファイルのFCBアドレスを返します．

3-5 バッファードファイルI／O関数

バッファードファイルI／Oはファイルと1文字単位でのデータのやりとりを可能にする非常に便利な関数の集りです．これらの関数を使う時は必ず"bdscio. h"をインクルードし，

```
FILE    iobuf;
```

などと宣言し，入出力に必要なバッファ領域を確保してから行ないます．このFILEというのは，bdscio.hの中で定義されており，

```
#define FILE struct _buf
```

となっています．バッファードファイルI／Oは，実際には前項の低レベルファイルI／O関数を用いて作成されていますので，この_buf構造の構造体変数iobufの中にファイル識別子，バッファへのポインタなどが設定されるわけです．

バッファードファイルI／Oは非常に便利な関数群ですが，残念ながら低レベルファイルI／O関数に比べると処理は遅くなります．もしどちらの関数を用いても余り作成の手間が変わらないのであれば，低レベルファイルI／Oを用いた方が処理は早くなります．一般にエラー時には－1を返します．

```
int fopen (filename, iobuf)
```

バッファードI／Oのため，ファイルをオープンします．

```
int getc (iobuf)
```

ファイルから1文字取り出します．エラー及びファイル終了時には－1を返します．EOFコード(1AH)もそのまま返します．

```
int ungetc (iobuf)
```

次のgetcで受け取る文字をプッシュバックします．

```
int getw (iobuf)
```

ファイルから16ビット値を取り出します．エラー及びファイル終了時には－1を返します．

```
int fcreat (filename,iobuf)
```

同じ名前のファイルがあれば消去し，ファイルをオープンします．

```
int putc (c,iobuf)
```

文字cをファイルに出力します．エラーの時は－1を返します．LFをCR－LFに変換しません．iobufとして，次のものは特殊な動作をします．

putc (c,1)	putchar(c)と同じです．
putc (c,2)	LST：(プリンター)にcを出力します．
putc (c,3)	PUN：(パンチャー)にcを出力します．
putc (c,4)	どのような場合でも，必ず画面にcを出力します．(DIOなどで使います)

```
int putw (w,iobuf)
```

16ビット値をファイルに出力します．エラー時には－1を返します．

```
int fflush (iobuf)
```

出力バッファにあるデータをディスクに書き込みます．

```
int fclose (iobuf)
```

ファイルをクローズします．fflushは自動的に行なわれまが，CP／Mのエンドオブファイルコード（1AH)は自動的に書き込みません．

```
int  fprintf(iobuf,arg1,arg2,…)
```

ファイルを出力対象とした，printfです．

```
int  fscanf(iobuf,format,arg1,arg2,…)
```

ファイルを入力対象としたscanfです．

```
char  *fgets(str,iobuf)
```

入力ファイルから1行読み込み，strに格納します．出力はstrになりますが，ファイル終了時には0を返します．

```
int  fputs (str,iobuf)
```

出力ファイルに文字列を書き出します．LFはCR，LFに変換します．

```
int  fappend (name,iobuf)
```

nameというファイルの終りから書き足すためにファイルをオープンします．

第 4 章

プログラムパッケージ

BDS　Cにはいくつかのユーティリティプログラムパッケージが附属しています．これらは標準関数と共に非常に便利なもので，うまく使うと何倍もプログラム作成が早くできます．

プログラムパッケージには，DIOなどのようにプログラム中に組み込み，新しい機能を追加するものと，CDB，L2のようにプログラム開発の手助けをするものとがあります．本章ではこれらについて詳細に解説します．

4－1　DIOパッケージ

DIOは説明せずに今迄に何回も利用してきましたが，Cにおける標準入出力（getchar，putchar）についてI／Oリダイレクションを実現するパッケージです．I／Oリダイレクションとは，入出力の方向をコマンドラインで指定したコンソールやファイルに切り換える機能で，例えば第2章で作成したプログラム“pp. c”では

```
A>PP
```

だけだと入力がコンソール(キーボード)，出力もコンソール（画面）であるため，タイプした文字が画面にそのままコピーされるという単純な2度打ちの機能しかありませんが，

```
A>PP  <PP.C
```

とすると，入力がコンソールではなくpp. cというディスク上のファイルに変わるので，pp. cの内容を画面上に表示します．TYPEコマンドと同じになるわけです．次に，

```
A>PP  >TEST
```

とすれば，キーボードからタイプした文字の内容をそのままディスクに書き込み，“TEST”というファイルを作成します．

```
A>PP  <PP.C  >TEST
```

では，pp. cをそのまま“test”というファイルにコピーします．

このように，＞と＜を用いて入出力を切り換えるのですが，この他に次のような文字が使えます．

＋　　＞と同じですが，ファイルに書き出すのと同じ内容を同時に画

面に表示します.

! パイプ機能を実現します．例えば,

```
A>PP <FILE1 !PP2 >FILE2
```

とすると，FILE 1をPPの入力とし，その出力をPP 2の入力とし，出力をFILE 2に書き出します.

これらの文字を使う時はその後のファイル名などとの間にスペースを入れないようにします.

DIOパッケージを利用する際は先にDIO.Cをコンパイルして DIO.CRL をディスクに作成しておき，メインプログラムとリンクするのが簡単です.

```
cc pp
clink pp dio
```

とします.

さて，DIO.Cの中では次の5つの関数が定義されています.

(1) dioinit

main関数の最初に実行します．コマンドラインを解釈し，入出力ファイルのオープンなどを行ないます．書式は

```
dioinit(&argc,argv);
```

です．argc. argvはmain関数への引数ですが，第1引数はargcのアドレスです．頭に&を付ける事を忘れないようにしてください.

(2) dioflush

プログラムが終了する時に必ず実行します.

(3)　getchar

DIO版のgetchar

(4)　putchar

DIO版のputchar

(5)　ungetch

DIO版のungetch

このうち，(3)～(5)は標準関数ですから，DEFF.CRL，およびDEFF2.CRLの中に本来の関数が存在しています．しかし，DIOをリンクした場合にはこちらが優先され，標準関数のファイルからgetcharなどの関数を捜してリンクする事はありません．

さて，プログラムの中で，I／Oリダイレクションの恩恵が受けられるのは，基本的にはgetchar，putcharの2つだけです．複雑なプログラムをこれだけで作成するのは大変なことですが，実際にはこれらの関数を利用している上位関数はすべて影響を受けます．文字，あるいは文字列の入出力機能を持つ標準関数のうち，DIOの機能が利用できる関数は次のとおりです．

```
getchar,printf,puts,ungetch,puchar
putc（引数が1のとき）
```

機能の変わらない関数は，

```
getline,gets,kbhit,lprintf,putch,scanf,
putc（引数が4の時）
```

です．printfが利用できますので，出力プログラムの作成時にはまず問題ありませんが，厄介なのは行単位で入力したい時です．getlineもgetsも使えま

せん．そこで，getcharを利用したgetlineをリスト4-1に作成しておきましたので，利用してください．ただし，次の点で通常のgetline関数と異なります．

(a) ファイル終了時にはEOF（－1）を返します．

(b) DIOがバッファードコンソールモードでコンパイルされている時はコンソール入力が最大135文字となります（getcharの内部でgets関数を利用しているためです）．

バッファードコンソールモードでは，最初のgetcharでリターンキーが押されるまでキーボード入力をバッファ内に取り込み，次のgetcharからはバッファ内から文字を拾ってきます．空になれば，再び最初から繰り返します．これはキーのとりこぼしを防ぐのに役立ちますが，BDS Cの標準関数とは動作が異なるため，問題になる時は“DIO.H”内のBUF_CONSのマクロ定義を1から0とし，DIO.Cを再コンパイルします．

DIOにはこのようなコンパイルスイッチがもう一つあります．これはABORT_CHECKで，putchar実行中にコントロールCによる中断を受け付けるか否かを指定します．1の時受付け，0で無視します．デフォルトは1です．

〈注〉BUF_CONSフラグはプログラム上ではデフォルト1ですが，プログラム内のコメントではデフォルトは0であると説明されています．BDS Cのバージョンによってはデフォルト0のものがあるのではないかと思います．

リスト4-1 DIO 版getlineサンプルプログラム

```
#include        <bdscio.h>
#include        <dio.h>

#define BUFMAX  135
char    buf[BUFMAX];
```

```
main(argc,argv)

int     argc;
char    **argv;

{
        int     c;
        int     line;
        line = 1;

        dioinit (&argc,argv);

        while  ( getline (buf,BUFMAX) != EOF)
               printf (" %4d:  %s¥n",line++, buf);

        dioflush ();
}
```

行番号をつけて出力するプログラム
PIP の〔n〕オプションと同じ。
ただし，135文字以上の行については2行以上に分割される．

```
getline( buf,len )
char    *buf;
int     len;
{
        int c, cnt;
        len -= 2;
        if ( (c = getchar() ) == EOF ) return ( EOF );

        cnt = 0;
        do
        {
               if ( c == '¥n') break;
               *buf++ = c;
               cnt++;
        }
        while (len-- && (c = getchar()) != EOF );

        *buf = '¥0';
        return (cnt);
}
```

int getline(buf,len)

buf　キーバッファの先頭アドレス
len　1行の最大文字数
出力文字数
　リターンのみの時：0
　最大文字数：len－1
　ファイル終了時：－1

4-2　WILDEXPパッケージ

WILDEXPパッケージはコンソールから入力したファイル名にワイルドマッチカードが使えるようにするものです．例えば，

```
A>SAMPLE *.C
```

とすると，＊.Cに相当するファイル名がコマンドラインにすべて並んだ状態に変換されて以降の処理を続けることができます．＊.Cに相当するファイルがSAMPLE.CとPP.Cの2つだったとすると，

```
A>SAMPLE SAMPLE.C PP.C
```

と入力されたのと全く同じになります．複数のファイルを扱う場合には非常に便利です．

このWILDEXPは次のように利用します．

```
main(argc,argv)
int     argc;
char    **argv;
{
        〈変数の定義〉
            :
    if(wildexp(&argc,&argv) == ERROR)
        exit();                 /*エラー処理*/
    dioinit(&argc,argv);        /*DIOを同時に使う時のみ*/
            :
}
```

WILDEXPでコマンドラインから入力できる記号は次の通りです．

- ＊　この記号以降の文字が何であっても一致したとみなします
- ？　？記号の文字が何であっても一致したとみなします
- !　上記の記号で一致したものの中で！の次にあるファイルを除きます

（例）　`A>SAMPLE *.* !SAMPLE.*`

このようにすると，SAMPLE.＊に一致するもの以外のすべてのファイル

名がコマンドラインに展開されたのと同様に扱えます．

WILDEXPを利用するためには，DIOと同じようにあらかじめ“wildexp.c”をコンパイルしておき，リンク時に指定して結合します．

WILDEXPのサンプル“dr.c”をリスト4-2に示しておきます．このプログラムはコマンドDIRと似たようなものですが，qsortを用いてabc順に並べ換えるため，少し見やすくなります．

リスト4-2　WILDEXPサンプルプログラムDR.C

```
#include        <bdscio.h>
#include        <dio.h>

#define ERROR   -1
#define _PNT    2       /* sizeof pointer              */
#define TABLEN  8       /* tabulation skip length      */

main( argc,argv )
char    *argv[];
int     argc;
{
        int     strcmp2 ();
        int     i,j;

        if ( wildexp ( &argc,&argv ) == ERROR )
                return ( puts ( "Wildexp overflow" ));
        dioinit (&argc,argv);

        qsort ( argv + 1, argc - 1, _PNT, strcmp2 );


        /****   type 4 files per line   *****/

        for ( i = 1; i < argc; i += 4 )
        {
                for ( j = 0; j < 4 && i+j < argc; j++ )
                {
                        printf ( "%s¥t", argv[i+j] );
                        if ( strlen ( argv[i+j] ) < TABLEN ) printf ("¥t");
                }
                printf ("¥n");
        }
        dioflush ();
}
```

```
/****   String compare through pointer   ****/


strcmp2( x,y )
char    **x, **y;
{

        return ( strcmp ( *x, *y ));
}
```

WILDEXP自身の問題でもありますが，dr.cでは，次の事に注意してください．

(1) デフォルトが設定されない

コマンドラインに何も書かれていない時には一切引数は与えられません．*.*をコマンドラインで与えない限り，“すべてのファイル”という指定は得られません．

(2) ファイルが存在するか否かのチェックができない

本当にコマンドラインで与えられたものなのか，ワイルドマッチカードによってディレクトリから読み取られたファイル名なのかわかりません．

例えば，

A>DR ABC.COM

とすると，ABC.COMと表示されますが，これはコマンドラインをそのまま写しただけで，DIRコマンドのようにファイルがあるかないかのチェックは行なわれていません．

これらは他のプログラム作成でも問題になる所だと思います．要するに，ディレクトリ検索用には使いにくく，むしろこのような場合には，WILDEXP

をコマンドライン展開用の関数と考えず，

```
int   i:
i=2;
wildexp(&i,"*.*");
```

という形で一般の関数として扱った方が使いやすいでしょう．

なお，wildexpで展開されるファイル名は200個まで（変更可能）で，ファイル名を格納するエリアは関数sbrkを用いてフリーエリアから確保されます．なお，sbrkでメモリが確保できなかった場合は－1(ERROR)を返します．

4－3　浮動小数点(float)パッケージ

浮動小数点パッケージは整数しか使う事のできないBDS Cで関数の形で浮動小数点演算を行なえるようにしたもので，技術計算などに威力を発揮します．ただし，三角関数，対数関数などはありませんので，高度な科学計算へ応用する場合はそれらの関数を自分で作成する必要があります．

浮動小数点変数は5バイトで表されます．従って一般的にはchar型変数配列を用いて，

```
char   flvar[5]; ………(A)
```

と宣言しますが，これでは浮動小数点変数である事が不明確なので，構造体を用いてプログラム先頭で

```
struct _float   {
        char fldim[5];
        };
#define float struct _float
```

と宣言しておくと，プログラム中では

```
float   flvar;       ………(B)
```

と浮動小数点変数を宣言できます．ただし，お気付きのように本来文字配列でなければならないものを構造体で宣言するのですから，厳密には誤った使い方となります．つまり浮動小数点パッケージ中の関数を

```
flfunc(flvar,……);
```

のように呼びだすと，文字配列として宣言した場合（A）は文字配列flvarの先頭アドレスが引数となり，構造体で宣言した場合（B）は構造体_float型の変数flvarへのポインタが引数となりますから，誤った型の使い方であり，物理的な数値としてたまたま同じだというだけなのです．

このように乱暴な方法はBDS C以外のCコンパイラでは受け付けない場合もあるでしょう．しかしながら，もともとfloat関数は他のC処理系と全く移植性のないものであり，BDS Cやα-C以外で使う事は考えられませんから，C言語らしくないと排除する必要もないでしょう．

それでは，簡単なプログラムを作成してみます．

操作例4-3　浮動小数点パッケージ

```
A>cc float  ←――― 浮動小数点ライブラリ float.c を先にコンパイルしておく．

BD Software C Compiler v1.50a (part I)
  30K elbowroom
BD Software C Compiler v1.50 (part II)
  28K to spare

A>type fptest.c
```

〈サンプルプログラム　fptest.c〉

```
#include        <bdscio.h>

struct  _float                      ┐
        {                           │
                char    fp[5];      │ float定義
        };                          │
                                    │
#define float   struct _float       ┘
```

```
main()
{
        int     i;
        float   a, b;

        atof ( a, "1.0" );◄──── 浮動小数点ライブラリでは，
                                 a=1.0をこのように実現する．
        atof ( b, "0.1" );

        for ( i = 0; i < 20; i++ )      ┌─ ここで用いている printf は
        {                               │  一般の printf 関数とは異なり，
                                        │  浮動小数点ライブラリにある
                printf ( "%4d= %f¥t", i, a );◄┘  printf である．
                fpadd ( a, a ,b );      /* a = a + b    */
        }
}
```

```
A>cc fptest

BD Software C Compiler v1.50a (part I)
  35K elbowroom
BD Software C Compiler v1.50 (part II)
  32K to spare


A>clink fptest -f float◄──── −f オプションを使うと，その後にあるファイルから
                             必要なファンクションだけを探し，リンクする．
                             特に printf が使われている時は，−f のあとに
BD Software C Linker   v1.50 float があると，float.crl の中から float 版の
Linkage complete             printf を探し，リンクする．
  39K left over


A>fptest
   0= 1.000000     1= 1.100000     2= 1.200000     3= 1.300000     4= 1.400000
   5= 1.500000     6= 1.600000     7= 1.700000     8= 1.800000     9= 1.900000
  10= 2.000000    11= 2.100000  ┌►12= 2.199999    13= 2.299999    14= 2.399999
  15= 2.499999    16= 2.599999  │ 17= 2.699999    18= 2.799999    19= 2.899999
                                └── 浮動小数点の常として，誤差がでてきます．
A>                                  プログラムを作成する時は十分注意してください．
```

以上でほぼおわかりだとは思いますが，float 演算はすべて関数の形で行なわれます．そして演算の多くは，

```
a = b + c
    ↓
func(a,b,c);
```

のように表されます．戻り値は文字配列 a の先頭アドレスとなります．

以下，パッケージ中に含まれる関数を説明します．

(1) 演算関数

```
fpadd  (result,op1,op2)
fpsub  (result,op1,op2)
fpmult (result,op1,op2)
fpdiv  (result,op1,op2)
```

それぞれ，op 1 と op 2 に対し，加算，減算，乗算，除算を行ないその結果を result に格納します．op 1，op 2，result とも浮動小数点数（5 バイトの文字配列）へのポインタで，戻り値は result へのポインタとなります．

(2) 変換関数

```
atof(op1,s1)
ftoa(s1,op1)
```

atof はアスキー文字列 s 1 を浮動小数点数に変換し op 1 に格納します．ftoa は浮動小数点数 op 1 をアスキー文字列に変換し，s 1 に格納します．

文字列の表現は，通常の小数によるほか，

1.2345e6 (1.2345 * 10の6乗)

などの指数表現が使えます．この場合，指数は－38から38までです．

```
itof(op1,n)
```

整数nの値を浮動小数点数に変換しop1に格納します．

(3) その他

```
fpcomp(op1,op2)
```

op1とopの値を比較します．戻り値は

```
op1>op2     1
op1<op2    -1
op1=op2     0
```

となります．

```
printf(format,arg1,arg2,…)
```

浮動小数点数の表示を可能とするfloat版printfです．標準関数のprintfの機能に加え，次のような変換文字を持ちます．

e	浮動小数点数を指数点数表現にします
f	浮動小数点数を小数点表現に変換します

従って，%fとする事により，浮動小数点数を画面に表示できます．%5.2fとすれば，整数部5桁，小数部2桁で表示します．ただし，丸めは行なわれないので，3.999という数値を%5.2fとしても表示は“3.99”となります．

4-4 倍精度整数（long）パッケージ

倍精度整数演算パッケージも，BDS Cのさらに高度な応用のために作成されたものです．浮動小数点数と異なり，扱える値の範囲は広くありませんが有効桁数が長く正確な値が求まりますから一般の整数（int）と併用して使う機会が多いでしょう．特にint同志の乗算の場合，正しく求まるのは結果が－32768～32767の場合で，200＊200程度の計算でオーバーフローしてしまいます．さらに都合の悪い事にプログラム中ではオーバーフローしたかどうかのチェックができません（longでは－2147483647～2147483647）．

そこで，この倍精度整数演算パッケージの登場になるわけです．

倍精度整数は，4バイトのchar型配列で表されます．floatと同様に構造体を用い，_longとでも宣言できるようにしておくのが便利でしょう（“long”は関数名なので使えません）．

使い方はfloatと同じですが，専用のprintf関数はありませんので倍精度変数の値をコンソールに出力する際は，ltoa関数（long型を数字の文字列に変換する関数）を使い，

```
char   s[12];
         ⋮
printf("−%s−",ltoa(s,longvar));
```

という形で使えば良いでしょう．ここで，sは変換した文字列を入れるバッファ（12バイト），longvarは表示したいlong型4バイトの先頭アドレスです．

サンプルプログラムとコンパイル例を操作例4-4に示します．この中に示されている関数muldivは

```
muldiv(a,b,c);
```

とすると，int型変数a，b，cについてa＊b／cを計算し，その結果を返します．中間演算をlong型で行ないますのでオーバーフローせず，正確な値が求

まります．int 同士の乗算ではちょっとした計算でもすぐにオーバーフローしますから，非常に便利な関数です．

操作例 4-4 倍精度整数パッケージのサンプル

```
A>cc long
BD Software C Compiler v1.50a (part I)
  34K elbowroom
BD Software C Compiler v1.50 (part II)
  30K to spare


A>type lgtest.c
```

〈サンプルプログラム〉 lgtest.c

```
#include      <bdscio.h>

struct  _lg
        {
                char     _lgdim[4];    } —long 定義
        };
#define _long    struct  _lg

main()
{
        int     a,b,c;

        a = 1000;
        b = 1001;
        c = 500;

        printf ( "calculate %d * %d / %d¥n",a,b,c );
        printf ( "int  result= %d¥n", a*b/c );
        printf ( "long result= %d¥n", muldiv (a,b,c));
}

muldiv(a,b,c)   /*   a*b/c      */ ← 有用な関数 muldiv
int     a,b,c;
{
        _long    la,lb,lc,ld;

        itol ( la, a );
        itol ( lb, b );
        itol ( lc ,c );
                              } 演算そのものは
                                倍精度で行なう．
        lmul ( ld, la, lb );
        ldiv ( ld, ld, lc );
```

```
        return ( ltoi (ld) );
}
```

```
A>cc lgtest
BD Software C Compiler v1.50a (part I)
  35K elbowroom
BD Software C Compiler v1.50 (part II)
  31K to spare

A>clink lgtest -f long
BD Software C Linker   v1.50
Linkage complete
  42K left over

A>lgtest
calculate 1000 * 1001 / 500
int  result= 35◄────────── 整数計算ではオーバーフローして誤り.

long result= 2002◄────────── 倍精度計算では正しい答え.

A>
```

倍精度整数パッケージに含まれる関数は次の通りです.

(1) 演算関数

```
ladd(result,op1,op2)
lsub(result,op1,op2)
lmul(result,op1,op2)
ldiv(result,op1,op2)
lmod(result,op1,op2)
```

long 型変数 op1, op2 に対し加算, 減算, 乗算, 除算, 剰余算を行ない, 結果を result に格納します. 戻り値はありません.

(2) 変換関数

```
atol(l,s)
char *ltoa(s,l)
```

atolはアスキー文字列sをlong変数に変換し，lに格納します．ltoaはその逆ですが，sをそのまま戻り値にします．lは倍精度整数文字配列の先頭アドレスです．

```
itol(l,n)
ltoi(l)
```

itolは整数nをlong変数に変換し，lに格納します．ltoiはlong変数lを整数に変換し，戻り値とします．

```
utol(l,u)
ltou(l)
```

utolは符号なし整数uをlong型に変換しlに格納します．ltouはlong型変数を符号なし変数に変換し戻り値とします．

(3) その他

```
lcomp(op1,op2)
```

long型変数op1とopの値を比較します．その状況によって，次のようになります．

```
op1>op2      1
op1<op2     -1
```

```
opl=op2      0
```

を返します．

```
char *lassign(dest,source)
```

long型変数sourceをlong型変数destに代入します．戻り値はdestへのポインタとなります．

4-5　α-Cでfloat，longを使う方法

α-Cには浮動小数点パッケージ，倍精度整数演算パッケージは附属されていない事になっています．しかし，実際には，そのように宣伝されているだけで，標準関数のCRLファイル（DEFF2.CRL）中にこれらの主要部分（アセンブラで書かれた部分）が組み込まれており，利用する事ができます．

このアセンブラ部はfp関数，long関数の2つしかありませんが，実はこの両者は機能コードを指定してコールすれば様々な処理を行なう多機能関数であり，パッケージ中の多くの部分がここに収められています．実際，浮動小数点パッケージ・倍精度演算パッケージの中には，単純にfp，long関数をコールするだけで，かえって呼び出し時の引数の設定がダブるため使わない方が有利なものまであります．fp，long関数の機能を表4－5，表4－6に示しておきます．

表4-5　fp関数機能表

機能コード	同等機能の関数	機　　能
0	fpnorm	浮動小数点数を正規化する
1	fpadd	浮動小数点数の加算を行なう
2	fpsub	浮動小数点数の減算を行なう
3	fpmult	浮動小数点数の乗算を行なう
4	fpdiv	浮動小数点数の除算を行なう
5	ftoa	浮動小数点数を文字列に変換する

〈注意〉

浮動小数点の場合，同一の値が何種類もの内部形式で表わされる事があります（0.1＊10，0.01＊100など）．従って，それを最も妥当な内部データに修正する事を正規化といいます．

表4-6　long関数機能表

機能コード	同等機能の関数	機　　能
0	itol	整数を倍精度整数に変換する
1	lcomp	倍精度整数の比較を行なう
2	ladd	倍精度整数の加算を行なう
3	lsub	倍精度整数の減算を行なう
4	lmul	倍精度整数の乗算を行なう
5	ldiv	倍精度整数の除算を行なう
6	lmod	倍精度整数の除算を行い，余りを求める

fp，long関数は，

```
fp(code,result,op1,op2);
long(code,result,op1,op2);
```

という書式で用います．result，op 1，op 2は共にfloat，あるいはlong変数

へのポインタです．

機能とコードをいちいち考えるのは大変なので，リスト 4-7 のようなヘッダーファイルを作成し，浮動小数点演算，倍精度整数演算を利用する際はプログラム先頭でインクルードするようにすれば良いでしょう．関数呼び出しではなく，マクロによる展開なのでこれらのパッケージを持っている BDS C ユーザーの方も，このヘッダーファイルを有効に生かせます．

リスト 4-7 fplong.h

```
/*      float macro function     */

#define fpnorm(op1)             fp(0,op1,op1)
#define fpadd(r,op1,op2)        (fp(1,r,op1,op2),r)      /* r <- op1+op2 */
#define fpsub(r,op1,op2)        (fp(2,r,op1,op2),r)      /* r <- op1-op2 */
#define fpmult(r,op1,op2)       (fp(3,r,op1,op2),r)      /* r <- op1*op2 */
#define fpdiv(r,op1,op2)        (fp(4,r,op1,op2),r)      /* r <- op1/op2 */
#define ftoa(r,op)              (fp(5,r,op),r)           /* float to ascii */

/*      long macro function      */

#define itol(r,op)              long(0,r,op)             /* int to long  */
#define lcomp(op1,op2)          long(1,op1,op2)          /* if (op1==op2) 0
                                                            op1>op2 ? 1:-1; */
#define ladd(r,op1,op2)         long(2,r,op1,op2)        /* r <- op1+op2 */
#define lsub(r,op1,op2)         long(3,r,op1,op2)        /* r <- op1-op2 */
#define lmul(r,op1,op2)         long(4,r,op1,op2)        /* r <- op1*op2 */
#define ldiv(r,op1,op2)         long(5,r,op1,op2)        /* r <- op1/op2 */
#define lmod(r,op1,op2)         long(6,r,op1,op2)        /* r <- op1%op2 */

struct  _float
        {
                char    _fpdim[5];
        };
#define float   struct  _float

struct  _lg
        {
                char    _lgdim[4];
        };
#define _long   struct  _lg
```

なお，fp関数の定義の中で，fadd関数などは

```
#define fpadd(r,op1,op2)
                 (fp(1,r,op1,op2),r)
```

と見慣れない定義になっています．fpadd( )は戻り値と型がr(esult)でなければなりませんが，fp( )は戻り値が規定されていないため，コンパチブルにするにはこのように定義をする必要があるのです．ここで用いているのはコンマ演算子といい，

```
(func( ),r)
```

とすると，func( )を実行し，式全体の値と型をrにします．これはfor文で最も多用されます．

なお，当然ですが，fp，long関数だけではサポートされない機能がいくつかあります(表4-8)．これらは必要に応じて作成する必要がありますが，容易に作成できるものもあれば，float版printfのように150行にもなるものもあります．作成の際の参考のため，両者の構造を図4-9，図4-10に示します．

表4-8　fp，long関数でサポートされていない機能

関数名	機能
fpcomp	浮動小数点の大小比較を行なう
atof	文字列 → 浮動小数点数の変換を行なう
itof	整数 → 浮動小数点数の変換を行なう
printf	浮動小数点版printf
ltoi	倍精度整数 → 整数の変換を行なう
atol	文字列 → 倍精度整数の変換を行なう
ltoa	倍精度整数 → 文字列の変換を行なう
lassign	倍精度整数の代入を行なう
ltou	倍精度整数 → 符号なし整数の変換を行なう
utol	符号なし整数 → 倍精度整数

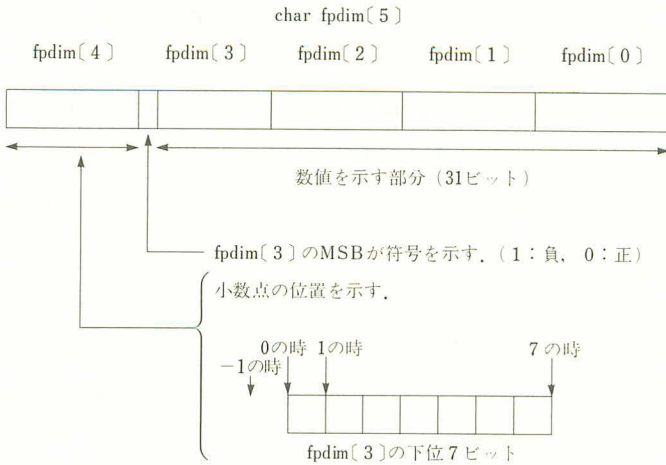



図4-9 浮動小数点数5バイトの構造

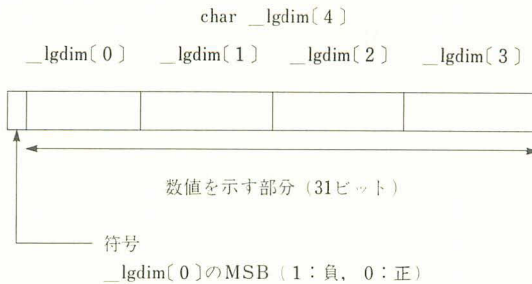



図4-10 倍精度整数4バイトの構造

4-6 CDBとデバッグ

C言語は非常に洗練され，自由にプログラミングがしやすい言語ですが，パソコンにおけるCは残念ながらデバッグしやすいとは言えません．用途によってはアセンブラより難しいと言われる事もあります．

これは8080，8085，Z80などではアセンブラが標準言語としてありとあらゆる開発ツールと実績（ノウハウ）が揃っているためです．満足のいく(?)

開発環境にある技術者にとっては裸のままのC言語ではとても使いやすいとは言えないのです．

「開発環境の整備」，これが，現在のC言語に与えられた最大のテーマでしょう．この目的のため現在多くの方法が試みられています．デバッグを目的としたインタプリンタや専用のシンボリックデバッガなどの開発がその良い例でしょう．

BDS Cは専用のシンボリックデバッガを用意している数少ないC処理系の1つです．完成度はお世辞にも高いとは言えませんが，CDBを利用する事で，開発効率は確実に何割かアップします．本項では一般的なCのデバッグ法と，CDBデバッガについて解説します．

なお，CDBはα-Cには附属していません．

(1) CDBとL2の作成

CDBパッケージはCP／Mのメモリレイアウトに依存する部分がありますので，最初のソースプログラムを一部修正してコンパイルし，実行可能なファイルを作る必要があります．この操作のため，BDS Cのマスターディスクには CDBGEN というプログラム自動作成サブミットファイルが附属しており（ライフボート版のみ），これを用いる事で簡単にCDBとL2を作成する事ができます．ただし，注意しなくてはならないのはこのサブミットファイルの実行開始時に指定するメモリアドレスです．マニュアルではDDTによって5番地を逆アセンブルすれば，そのジャンプアドレスがBDOSのエントリであるとされていますが，これは誤りです．この値を用いても正しく動作するものが作成できますが，CDBもDDTなどと同じようにメモリの上位番地にターゲットプログラムと同居する形をとりますから，なるべく上位ぎりぎりに設定した方が大きなターゲットプログラムがデバッグできるようになります．自分のCP／MシステムのBDOSの先頭アドレスを知る一番簡単な方法は，電源をいれてCP／Mがスタートした時に出る，“○○K CP／M”というオープニングメッセージを見る事です．このメッセージと表4-11を参照す

れば，BDOSのアドレス，及び引数として与える値が求まります．

このサブミットファイルは意外にトラブルが多く，引数を誤ったり，文字を正しく書かないと初めからやり直さなければなりません．かなり時間のかかる操作（10～30分位）ですからやり直しは大変です．失敗のないように気を付けてください．

なお，5インチディスクの場合はCDBGEN5.SUB，8インチの場合はCDBGEN8.SUBを使います．これで，CDB.COMと本体CDB2.OVLの2つとリンカL2.COMが自動的に作成されます．

表4-11　CP／Mのメモリサイズと CDBGENの引数の関係

CP／Mサイズ	BDOSアドレス	第1引数	第2引数	機　種　例
64K	EC06	9800	DE00	PDS－7
63K	E 806	9400	DA00	QC－10
62K	E 406	9000	D600	
61K	E 006	8C00	D200	
60K	DC06	8800	CE00	X1シリーズ
59K	D 806	8400	CA00	
58K	D 406	8000	C600	FP－1000
57K	D 006	7C00	C200	
56K	CC06	7800	BE00	PC，FMシリーズ
55K	C 806	7400	BA00	

〈注意〉

機種名はあくまで目安です．これ以外のものが発売されている可能性もありますので，必ず確認してください．

（2）Cの基本的なデバッグ法

Cの基本的なデバッグ法は他の高級言語と同じです．正しく動作するまでプログラムを修正し，コンパイルとリンクを繰り返します．BASICなどのように実行→修正が即座にできるわけではありませんし，デバッグ中に暴走

する事もあります．その場合はシステムから立ち上げ直さねばなりません．

デバッグのために最も良く用いられるのは“スタブ”です．スタブとはプログラムを実行した時にその中間結果を確かめるために，簡単なプログラムを挿入する方法で，例えば，

```
printf("x=%d,y=%d¥n",x,y);
```

などと所々に入れておけば，変数x，yの値の変化をチェックする事ができます．特にC言語の場合には＃if，＃else，＃endifという条件コンパイルの指示命令がありますから，これを利用します．

```
#define   DEBUG   1
              :
#if   DEBUG
        printf("x=%d,y=%d¥n",x,y);
#endif
```

などという形で使うわけです．このようにしておけば，プログラム完成時にDEBUGの設定を0にしさえすれば，自動的にコンパイル時にソースリストから除いてくれます．

スタブはCDBを使う場合でも併用される事が多いのですが，この方法の弱点はプログラムの内容によっては全く使えない場合があるという事です．アセンブラやマクロプロセッサのように画面表示を行なわず，ひたすら入力ファイルを読み，出力ファイルを書き出すようなプログラムでは問題ありませんが，きちんとした画面出力や，リアルタイムの制御が必要なプログラムでは，この方法は部分的にしか使えません．

ただ，これはあらゆる高級言語に言える事で，そのような目的のためには特別なデバッグ法を考える必要があります．

しかし，スタブは一般にかなり効果的なデバッグ法で，処理内容・ループの回数，実行そのものの有無を容易に判定できます．大きなプログラムにな

ってくると修正したあと再コンパイル，再リンクに時間がかかりますが，ひたすら耐えるしかありません．

(3) CDBの使い方

CDBの基本思想はトップダウンデバッグです．一般にアセンブラなどでは，設計はトップダウンでも，デバッグは下位のサブルーチンからボトムアップ方式で行なう事が多いので，異和感を持つ方があるかもしれません．

CDBでは，デバッグはすべてのプログラム作成後に行なうのが前提になっています．従って，基本的な関数をデバッグしたい場合や，全体が未完成で一部分だけをテストしたい時にはその関数を呼び出すプログラムを別に作成してから行ないます．この方法もプログラムが大きい場合にはかなり効果的です．

さて，CDBを用いる場合にはコンパイル，リンクは次のようにします．

```
A>cc prog -k
A>l2 prog -d
```

ccにオプション－k を付けるとCRLファイルを作ると同時に“.CDB”という拡張子を持つ変数名などのシンボルファイルを作成し，オブジェクト中にプログラムトレース用のリスタート命令を挿入します．リンクにはL2を使い，－dオプションを指定します．このオプションを指定すると，関数の先頭にリスタート命令を挿入し，関数のエントリアドレスのシンボルファイルを作成します．－d以外に－s，－nsというCDB用のオプションも用意されており，－d と組み合せて用います．

－s　このオプション以降にあるファイルはシステム関数として扱い，トレースしない事を指定します．CDBによるデバッグ中には実行がかなり遅くなりますので，積極的に－s オプションを指定した方が効率的です．

－ns　－s と逆にシステム関数でもトレースを行ないたい場合に使います．

以上のようにコンパイル，リンクを行なった後 CDB の起動は，

```
A>CDB prog
```

として行ないます．これで CDB は prog プログラムデバッグのために必要なシンボルファイルを読み込んで prog の実行を開始し，main 関数の先頭でブレークします．

CDB の起動時には次のようなオプション指定が可能です．

－l　ローカル変数ファイル（.CDB）のファイル名を指定します．分割コンパイルした場合に指定します．

－g　外部変数シンボルファイルのファイル名を指定します．一般にはすべてのソースファイルで同じ外部変数を指定しなければならないので，使う必要はありません．

－d　CDB2.OVL をロードするドライブを指定します．

%　この文字以降の文字列すべてをデバッグ用プログラムへコマンドライン引数として渡します．

CDB の説明については，コマンドの内容をくどくど説明するよりも，実際の使用例を示す方が判りやすいと思います．CDB パッケージの中には，そのために“target.c”というプログラムが附属しているのですが，比較的短いプログラムではあるものの，CP／M についてかなり知識がないと理解できないので，余りサンプルとして適当ではありません（プログラムとしては，非常にユニークで実用的なユーティリティです）．

そこで，私がサンプルとして作成したのは，CDB を使う際に便利なユーティリティで，関数ごとに行ナンバーをつけるプログラムです．CDB ではブレ

イクポイントを指定する際に，関数別の行ナンバーを必要とします．これは当然の事で，複数のプログラムファイルをリンクできるBDS Cではファイル中の絶対的な行番号は何の役にも立ちません．このプログラムは短く仕上がっていますので，サンプルとしては好都合でしょう．

さて，関数ごとに行番号をつける事になると，コンパイラと同じような構文解析をしなければなりません．本格的にこれを行なうのはかなり大変なので，ここでは大雑把なアルゴリズムを使う事にします．

1. カーリーブレイス（{}）の外側で，
2. 行中に“（“と”）”を持ち，
3. 文末記号（；）を持たない

ような行を，関数名を記述している行と判断します．この方法では，

```
#define itol(r,op) long(0,r,op)
```

という行でも関数名と判断してしまいます．しかし，マクロ宣言は通常関数外で行ないますから問題にはなりません．また，本来C言語の記述はフリーフォーマットですから，関数名の記述を

```
func(a) char a; {
          ⋮
```

のように1行中に引数まで書いてしまうと判定を誤ります．コメントは判定しますので，コメント中には何を書いても良いのですが，文字列（“ ”）および文字（‘ ’）の中は判定を省略していますので，文字の中に

```
{ , },/*,*/
```

があってはいけません．α-Cに含まれるソースファイルすべてをテストしてみましたが，問題になるファイルは1つだけでした．この制約を知ってさえいれば問題になる事は全くないでしょう．

なお，コメントはネストします．BDS Cで－cオプションを付けなければコンパイルできないファイルには正しい行番号を付ける事はできません．リスト4-12にこのサンプルプログラム“cnm.c”を示します．行番号はcnm自身によって発生させたものです．

リスト4-12　cnm.c

```
 1: /*****************************************************
 2:
 3:          Line Number Maker for CDB
 4:
 5:          Author: Tsu.Mitarai     12/oct/1985
 6:
 7:          A>cc cnm -e2800
 8:          A>clink cnm dio
 9:
10: *****************************************************/
11:
12: #define DEBUG    0
13:
14: #include         <bdscio.h>
15: #include         <dio.h>
16:
17: #define ERROR    -1
18: #define TRUE     1
19: #define FALSE    0
20:
21: char     lbuffer[512];    /* line buffer          */
22:
23: FILE     fpin;            /* input file           */
24:
25: char     *cpnt;           /* pointer to lbuffer   */
26: char     krflag;          /* ')' flag             */
27: char     klflag;          /* '(' flag             */
28: char     scflag;          /* semicolon flag       */
29: char     comcnt;          /* comment flag         */
30: int      lcnt;            /* line counter         */
31: int      brcnt;           /* curly brace counter  */
32:
33:
 1: main(argc, argv)
 2: int      argc;
 3: char     *argv[];
 4: {
 5:          if ( argc == 1 )
 6:                   exit (printf ("Usage: cnm infile [ >outfile ]"));
```

（12行目注：1にするとスタブが有効になる．）

（6行目注：コマンドのフォーマット表示．このようにしておくと使いやすい．）

```
 7:         dioinit (&argc, argv);
 8:         if ( ERROR == fopen (argv[1], fpin))◄──────ファイルのオープン.
 9:                 exit (printf ("Can't open %s", argv[1]));
10:
11:         cdbnum ();              /* main routine */
12:         dioflush ();
13: }
14:
15:
 1: cdbnum()
 2: {
 3:         char    c;
 4:         lcnt = 1;
 5:         brcnt = 0;
 6:         while ( fgets (lbuffer, fpin) )◄──────ファイルからの行入力.
 7:         {
 8:                 cpnt = lbuffer;
 9:                 krflag = FALSE;
10:                 klflag = FALSE;
11:                 scflag = FALSE;
12:
13:                 while (c = *cpnt++)
14:                 {
15:                         if (comcnt == 0)◄──────コメント中か否かのチェック.
16:                         {
17:                                 checkln (c);◄──────コメント外の処理.
18:                         }
19:                         else if (c == '*')         ┐
20:                         {                          │
21:                                 if (*cpnt == '/')  │ コメント終了をチェックし,
22:                                 {                  ├ 終了ならコメントカウンタを
23:                                         cpnt++;    │ デクリメント.
24:                                         comcnt--;  ┘
25:                                 }
26:                         }
27:                         else if (c == '/')         ┐
28:                         {                          │
29:                                 if (*cpnt == '*')  │ コメント中さらにコメント開始の／＊を
30:                                 {                  ├ チェックし,
31:                                         cpnt++;    │ あればコメントカウンタをインクリメント.
32:                                         comcnt++;  │
33:                                 }                  ┘
34:                         }
35:                 }
36:                 if      ((brcnt == 0) && (klflag == TRUE) &&       ┐ カーリーブレイスの外で,
37:                          (krflag == TRUE) && (scflag == FALSE))    ├ 行中に“(”と“)”があり,
38:                                                                    │ かつセミコロンがない時には
39:                         lcnt = 1;                                  ┘ ラインカウンタを初期化する.
```

```
40:
41: #if DEBUG
42:         printf ("com=%d,kl=%d,kr=%d,sc=%d,br=%d¥t",comcnt,klflag,krflag,scflag,brcnt);
43: #endif
44:
45:                 printf ( " %4d:  %s", lcnt++, lbuffer );
46:         }
47:         if ( brcnt )
48:                 printf (" -Unmatched right brace.");
49: }
50:
51:
 1: checkln(c)
 2: char    c;
 3: {
 4:         switch (c)
 5:         {
 6:         case '/':
 7:                 if(*cpnt == '*')
 8:                 {
 9:                         cpnt++;
10:                         comcnt++;
11:                 }
12:                 break;
13:         case '{':
14:                 brcnt++;
15:                 break;
16:         case '}':
17:                 if ( --brcnt < 0 )
18:                 {
19:                         printf (" -Unmatched left brace.");
20:                         exit();
21:                 }
22:                 break;
23:         case '(':
24:                 klflag = TRUE;
25:                 break;
26:         case ')':
27:                 krflag = TRUE;
28:                 break;
29:         case ';':
30:                 scflag = TRUE;
31:         default:;
32:         }
33: }
```

- 41〜43行：スタブ DEBUGが0になると無効になる．
- 45行：行番号と画面表示．
- 48行：エラー時の表示．
- 7〜12行：コメント開始のチェック．
- 13〜15行：カーリーブレイスのカウント．
- 16〜21行：カーリーブレイスのカウント．
- 23〜25行："("のチェック．
- 26〜28行：")"のチェック．
- 29〜31行：セミコロンのチェック．

それでは，このcnm.CをサンプルにCDBを使ってみましょう．

まず，CDBのコマンドを表4-13に示します．

表4-13 CDBのコマンド一覧表

CDBのコマンド書式	機　　　能
b[reak] [関数名][行番号 [パスカウント]]	ブレイクポイントの設定
r[eset] [関数名] [文番号]	ブレイクポイントの解除
clear	すべてのブレイクポイントの解除
g[o]	プログラムをブレイクポイントまで実行する
t[race] [文の数]	関数名，文番号を表示して実行
u[ntrace] [文の数]	表示を行なわないtrace
d[ump] 式 [データ数] [データ型]	変数，メモリ内容の表示
s[et] 変数名データ [c]（cはバイト時）	変数，メモリ内容のセット
l[ist]	現在の関数名とメモリ内容の表示
l[ist] a[rguments]	関数の引数の内容を表示する
l[ist] l[ocals]	ローカル変数の内容を表示する
l[ist] g[lobals]	外部変数の内容を表示する
l[ist] b[reakpoint]	設定されているブレイクポイントの表示
l[ist] m[ap]	関数のエントリアドレスをすべて表示する
l[ist] t[raceback]	関数呼び出しの履歴を表示する
run	プログラムをリアルタイムに実行する
quit	CDBを終了しCP/Mシステムに戻る
h[elp]	CDBのコマンド一覧を表示する

〈注意〉
変数名はアドレスで指定する事もできます．〔 〕内は省略可．

CDBによるデバッグ作業では，ブレイクポイントの設定 → 実行 → ブレイク，という形で，プログラムを少しずつ実行していきます．このブレイク

ポイントの設定はbreakコマンドで行ないますが，

```
b[reak] 関数名 [文番号 [パスカウント]]
```

とします．コマンドは“b”だけで構いません．文番号は，関数ごとの行番号（cnmによって発生する行番号）です．行番号を省略すると関数の先頭（0でも良い）となり，－1を指定すると関数終了時にブレークがかかります．

実行はgコマンドで行ないます．

```
g[o]
```

で現在中断している場所から実行を開始し，ブレイクポイントでブレイクします．また，途中でキーボードを押すとそこで実行を中断し，コマンド待ちになります．ただし，デバッグ中のプログラムでキーの入力待ちのときには中断できません．

次にdumpコマンドは

```
d[ump] 式 [データ数] [データ型]
```

となりますが，式の所は殆どの場合変数名を使いますが，アドレス（16進数の場合は0x を頭につける）も利用できます．データ型は自動的に宣言時の型が指定されますが，

c	文字
p	ポインタ
i or w	整数／word
s	文字列

を付けるとその型で表示します．

サンプルプログラムのCDB使用例を操作例4-14に示します．

操作例 4-14 CDBの使い方

```
A>cc cnm -k
BD Software C Compiler v1.50a (part I)
  33K elbowroom
BD Software C Compiler v1.50 (part II)
  30K to spare

A>l2 cnm dio -d
Mark of the Unicorn Linker ver. 2.2.2
Loading  CNM.CRL
Loading  DIO.CRL
Scanning DEFF.CRL
Scanning DEFF2.CRL

Link statistics:
  Number of functions: 41
  Code ends at: 0x2547
  Externals begin at: 0x2547
  Externals end at: 0x3411
  End of current TPA: 0xDC06
  Jump table bytes saved: 0x102
  Link space remaining: 18K
```

cnmの
コンパイルとリンク.

```
A>cdb cnm
c debugger ver 1.2
top of target stack is 8311, cdb2 is at 8800
globals use 01F0 bytes, locals use 004C bytes

break at MAIN 0 [08BB]

>g
Usage: cnm infile [ >outfile ]
```

引数を与えずに
実行してみる.
Usage：を表示して終了する.

```
A>cdb cnm %cnm.c
c debugger ver 1.2
top of target stack is 8311, cdb2 is at 8800
globals use 01F0 bytes, locals use 004C bytes

break at MAIN 0 [08BB]

>l m
MAIN      08B8  CDBNUM   0972  CHECKLN  0B4E  DIOINIT  0C42
DIOFLUSH  1045  GETCHAR  114D  PUTCHAR  125C  UNGETCH  1379
PRINTF    1398  FOPEN    13B5  FGETS    1402  FPUTS    14F9
STRCMP    1560  FCREAT   15F2  FCLOSE   1651  PUTC     1686
FFLUSH    17B1  GETC     1912  STRLEN   19E1  _SPR     1A21
UNGETC    1E0B  _USPR    1E79  ISDIGIT  1F1B  _GV2     1F4A
TOUPPER   1FA7  ISLOWER  1FDB  EXIT     200A  UNLINK   2010
```

A>cdb cnm %cnm.c ← cnm.c自身を指定してテストする.

>l m ← プログラム中にある関数名とエントリアドレスをすべて表示する.

```
RENAME    203A    EXECV     20A9    GETS     20E4    BDOS     2118
OPEN      212C    CREAT     2194    CLOSE    21CD    WRITE    2100
MOVMEM    2284    SEEK      22D5    READ     2320    EXECL    23C9
CFSIZE    24F5

>b cdbnum ←──────────── 関数cdbnumの先頭にブレークポイントをセットする.
>g ←──────────── 実行する.

break at CDBNUM 0 [0975] ←──────────── cdbnumの先頭でブレーク.

>b cdbnum 46 ←──────────── cdbnum 46行めにブレークポイントをセット.
>g
    1:  /****************************************************←──── プログラム実行による
                                                                   表示.

break at CDBNUM 46 [0B07]
                         ┌─── lbuffer(行入力バッファ)の内容を見る.
                         │    最後のSはstringの指定で，文字列として表示させる.
>d lbuffer s ←───────────┘
2E00 (len 53): "/****************************************************¥n"

>d lcnt ←──────────── lcnt(行番号カウンタ)の内容を見る.
[340D] = 0002 =      2  '..' ←──────────── すでにインクリメントされて2になっている.

>l b ←──────────── ブレークポイントのリストを表示.
CDBNUM 0
CDBNUM 46
MAIN -1

>clear ←──────────── ブレークポイントをすべてクリア.
>b cdbnum 39 ←──────────── cdbnum 39行めにブレイクポイントをセット，新しい関数の
                           定義がはじまるとブレークする.
>g
    2:                                                  ┐
    3:          Line Number Maker for CDB               │
    4:                                                  │
    5:          Author: Tsu.Mitarai     12/oct/1985     │
    6:                                                  │
                   ¦                                    │
                   ¦ (中略)                             ├ 正しく実行される.
                   ¦                                    │
   29:  char    comcnt;         /* comment flag       */│
   30:  int     lcnt;           /* line counter       */│
   31:  int     brcnt;          /* curly brace counter */│
   32:                                                  │
   33:                                                  ┘

break at CDBNUM 39 [0AD3]←──────────── ブレーク

>d lbuffer s
2E00 (len 17): "main(argc, argv)¥n"←──────────── ブレークした行は確かに新しい関数の
                                                  定義関始行.
```

```
>b cdbnum 46
>g
    1:  main(argc, argv)        } 行を表示させる.

break at CDBNUM 46 [0B07]

>r cdbnum 46 ◄──────── 設定したブレークポイントの解除.
>g
    2:  int     argc;
    3:  char    *argv[];
    4:  {
    5:          if ( argc == 1 )
    6:                  exit (printf ("Usage: cnm infile [ >outfile ]"));
    7:          dioinit (&argc, argv);
    8:          if ( ERROR == fopen (argv[1], fpin))
    9:                  exit (printf ("Can't open %s", argv[1]));      } 実行
   10:
   11:          cdbnum ();              /* main routine */
   12:          dioflush ();
   13:  }
   14:
   15:

break at CDBNUM 39 [0A03] ◄──────── 次の関数の先頭でブレーク.

>clear ◄──────── 全ブレークポイントの解除
>g
    1:  cdbnum()
    2:  {
    3:          char    c;
    4:          lcnt = 1;
    5:          brcnt = 0;
    6:          while ( fgets (lbuffer, fpin) )
    7:          {
                    ¦
                    ¦ (中略)
                    ¦
   48:                  printf (" -Unmatched right brace.");
   49:  }
   50:
   51:
    1:  checkln(c) ◄──────── 関数の頭で行番号が1に
    2:  char    c;                戻っている.
    3:  {
                ¦
                ¦ (中略)
                ¦
   31:          default:;
   32:          }
   33:  }
```

```
MAIN returning 2500 = 9472 = '%.'

>quit ←―――――――― CDBの終了

A>
```

また，このプログラムにはスタブが設定されています．スタブを有効にして実行した例を操作例4-15に示します．このプログラムの場合はスタブでデバッグした方が簡単でしょう．

操作例4-15　スタブによるデバッグ

```
com=1,kl=0,kr=0,sc=0,br=0          1: /*****************************************************
com=1,kl=0,kr=0,sc=0,br=0          2:
com=1,kl=0,kr=0,sc=0,br=0          3:         Line Number Maker for CDB
com=1,kl=0,kr=0,sc=0,br=0          4:
com=1,kl=0,kr=0,sc=0,br=0          5:         Author: Tsu.Mitarai      12/oct/1985
com=1,kl=0,kr=0,sc=0,br=0          6:
com=1,kl=0,kr=0,sc=0,br=0          7:         A>cc cnm -e2800
com=1,kl=0,kr=0,sc=0,br=0          8:         A>clink cnm dio
com=1,kl=0,kr=0,sc=0,br=0          9:
com=0,kl=0,kr=0,sc=0,br=0         10: *****************************************************/
com=0,kl=0,kr=0,sc=0,br=0         11:
com=0,kl=0,kr=0,sc=0,br=0         12: #define DEBUG   1

                                       ┆（中略）
                                       ┆

com=0,kl=1,kr=1,sc=0,br=0          1: main(argc, argv)
com=0,kl=0,kr=0,sc=1,br=0          2: int     argc;
com=0,kl=0,kr=0,sc=1,br=0          3: char    *argv[];
com=0,kl=0,kr=0,sc=0,br=1          4: {
com=0,kl=1,kr=1,sc=0,br=1          5:         if ( argc == 1 )
com=0,kl=1,kr=1,sc=1,br=1          6:                 exit (printf ("Usage: cnm infile [ >outfile ]"));
com=0,kl=1,kr=1,sc=1,br=1          7:         dioinit (&argc, argv);
com=0,kl=1,kr=1,sc=0,br=1          8:         if ( ERROR == fopen (argv[1], fpin))
com=0,kl=1,kr=1,sc=1,br=1          9:                 exit (printf ("Can't open %s", argv[1]));
com=0,kl=0,kr=0,sc=0,br=1         10:
com=0,kl=1,kr=1,sc=1,br=1         11:         cdbnum ();              /* main routine */
com=0,kl=1,kr=1,sc=1,br=1         12:         dioflush ();
com=0,kl=0,kr=0,sc=0,br=0         13: }
```

- br ― カーリーブレイスのカウンタ．
- sc ― セミコロンフラグ（行中にセミコロンがあれば1）．
- kr ― “)”のフラグ（行中にあれば1）．
- kl ― “(”のフラグ（行中にあれば1）．
- com ― コメントカウンタ（コメント中であれば1）．

CDBの問題点

CDBの使い方は今まで述べてきた通りですが，使う際にいくつか問題なる点があります．これらを良く知っておくと，デバッグの方針を立てる際に役立ちます．

(a)　関数単位でのデバッグができません．

特定の関数だけをデバッグしたい時は，その関数を何度も呼び出すようなテストプログラムを別に作成する必要があります．

(b)　実行の再現，後戻りができません．

実行中に発生した不具合はいつ，どのように発生したか再現させる事が困難です．もう一度，頭から注意深くテストしてみる必要があります．

(c)　ターゲットプログラムにバグがあると，暴走する事があります．

これは，CDBというより，CP／M自身の本質的な問題です．BDOS，BIOSなど，CP／Mのシステムを破壊してしまうと暴走し，システムの再立ち上げになります．

(d)　上位関数のローカル変数を見る事はできません．

CDBで参照できるのは外部変数と実行中の関数自身のローカル変数だけです．

(e)　プログラムを書き換える事はできません．

プログラムの変更は一切できません．追加，変更ではなく，削除だけでもできれば便利なのですが……．

(f)　CP／Mで動作するプログラムしかデバッグできません．

ROM化するようなプログラムにはCDBは利用できません．このような場合については，第7章を参照してください．

4-7 L2の使い方

L2はBDS Cで作成されたリロケーブルファイルをリンクするためにCLINKのかわりに作られた汎用リンカーです．機能的にもCLINKをほぼ包含しており，リンク速度はやや遅いものの，得られるオブジェクトが若干小さくなる所からCLINKよりも優れているといえるでしょう．

CLINKと大きく異なるのは，CDBと組み合せて使う事ができる点ですが，オプションなどの指定方法も異なっています．

－f　リンクする関数のテーブルサイズを変更します．オプションの後に10進数で指定します．デフォルト値は200です．

－l　このオプション以降のファイル名をライブラリファイルとして扱い，必要なものだけを抜き出してリンクします．（CLINK －fと同じ）

－m　プログラムがmain関数以外からスタートするように－mの後にその関数名を記述します．

－t　CLINKの－tと同じです．

－w　関数名のシンボルテーブルを出力します(SYMファイル)．

－wa　SYMファイルにリンク一覧表を追加する．

－ws　SYMファイルを出力し，リンク一覧表を．LNKファイルに出力する．

－ovl　name　addr

オーバーレイセグメントを作成する時に使います．addrにプログラムのベースアドレス，nameにはルートセグメントの作成時に作られたSYMファイル名です．

－org　addr

オーバーレイプログラムのルートセグメントを作成する時にベースアドレス（プログラム開始アドレス）を指定します．

CLINKの－ l と同じです.

なお，オプションを与えるときにはその後に続くアドレス，ファイル名などの前にスペースを入れます．CLINKでは不要な場合もあります．

L 2には，CLINKにないオプションがある半面，ないオプションもあります．多くの場合L 2の方が便利ですが，CLINKの－n（ノーブート），－z（外部変数のクリア禁止）などはありません．注意してください．

なお，オプションの問題とは別にリンクアルゴリズムそのものが異なっているため，作成されたオブジェクトが基本的に異なります．余り気にする必要はありませんが，一応ここで紹介しておきます．

(1) L 2でリンクすると外部関数参照時のジャンプテーブルが削除されます．そのため，プログラムが数％小さくなり実行も僅かに速くなります（第6章参照）.

(2) CLINKでは最初にリンクするファイルにmain関数がなければなりません．これは，CLINKが最初のファイルからmain関数を捜し，その関数モジュールをC.CCC（ランダムパッケージ）の直後に配置するためです．L2では，これを行なわず，ファイル中にある関数モジュールをそのままの順番でリンクし，C.CCCの中にあるmain関数へのジャンプベクターを書き換える方法をとっています．そのため，最初にリンクするファイル中にmain関数がなくても構いません．

この事については，第7章でも触れますが，BDS Cで作成したプログラムをCP／M以外の環境で動作させたい場合など，知らないと不便な場合があります．

(3) L 2とCLINKでは，リンクする際にCRLファイルのチェックの厳しさが異なります．これは，第5章のZCASMを作成していて気付いて気付いた事ですが，CLINKでは，指定ファイルをすべて取り込みますが，L 2ではCRLファイル中にあるファイルの長さを見てファイルをロード

します．ZCASMのようにじかにCRLファイルを操作するプログラムを作成する場合は，これを考慮してください．

(4) L2でリンクした場合には一部コードの書き替えを伴います．一般に関数のアドレスを参照する場合，BDS Cは外部関数へのジャンプテーブルからそのジャンプベクターを参照するコードを発生しますが，L2ではこのテーブルそのものを削除してしまうため，コード中にじかに関数のアドレスが埋め込まれるように変更されます．

第 5 章

BDS Cとアセンブラとのリンク

C言語はアセンブラに近い自由な記述が可能な言語ですが，特別な制御を行ないたい場合や，高速な処理が必要な時は，どうしてもアセンブラでプログラムを作成しなければなりません．とりわけ，CP/M以外のOS上で動作するプログラムや，ROM化するプログラムを作る場合には不可欠と言っても良いでしょう．

BDS CにはCASMと名付けられたプリプロセッサがC言語のソースリストで供給されており，このプログラムにより，アセンブラ言語で書かれたプログラムをBDS Cのリンカー（CLINK，L2）でリンクできる形式に変換する事ができます．

本章ではCASMの使い方と，BDS Cとリンクする事を目的としたプログラムの書き方を説明し，ザイログニーモニック版のZ80用アセンブラプリプロセッサ（ZCASM）を紹介します．CASMはα-Cにはありませんので，ZCASMを用いれば，α-Cのユーザーの方でもアセンブラが使えるようになります．

5-1　CASMの使い方

CASMはアセンブラで書かれたソースプログラムをCP/MのASMコマンドでアセンブルした時に，BDS Cのリロケータブルファイル（CRLファイル）形式になるように変換するプログラムです．

CASMはさほど難しくありませんが，BDS Cとのインターフェースや，その他いくつか注意しなければならない点があります．

サンプルでは，16ビット値をローテートする関数を作ってみましたが，プログラムそのものの書き方よりも，CASMを利用する手順の方が面倒なのでまず，理解しておかねばなりません．

操作例5-1　CASM使用法

A>type rrotate.csm

アセンブラ定義関数のサンプル

```
;       rrotate( x )
;       unsigned        x ;

        FUNCTION        rrotate

        pop     d       ;return address pop
        pop     h       ;arg
        push    h

        mov     a,l
        rar
        mov     a,h
        rar
        mov     h,a
        mov     a,l
        rar
        mov     l,a

        push    d       ;retrurn address push
        ret

        ENDFUNC
```

→ FUNCTIONは関数名を指定し，ここからCRLファイルのフォーマットに変換する事を指示する．

（このプログラムは，引数を右に1回ローテートするもので，Cではシフトしかできないので同様の処理は若干面倒になる．）

→ 関数の終了を指示．

```
A>casm rrotate
```
← rrotate.csm を casm により rrotate.asm に変換する，

```
BD Software CRL-format ASM Preprocessor v1.50
Processing the RROTATE function...
RROTATE.ASM is ready to be assembled.
```

```
A>asm rrotate
```
← rrotate.asm をアセンブル．

```
CP/M ASSEMBLER - VER 2.0
S0003 =          SECTORS$ EQU ($-TPALOC)/256+1 ;USE FOR "SAVE" !.
010C
000H USE FACTOR
END OF ASSEMBLY
```

このエラー行は必ず出力される．これは次のDDTのあと，マニュアルでSAVEを行わねばならないので，その引数を画面に表示するためである．
「S0003＝」なら，3である．

```
A>ddt rrotate.hex
```
← DDTにより，HEXファイルをバイナリ形式に変換する．LOADコマンドはエラーチェックがあるため使えない．

```
DDT VERS 2.2
NEXT  PC
0317 0000
-g0
```

```
A>save 3 rrotate.crl
```
← 引数の3はASMのエラー行で得られた値．

```
A>type test.c
```
← ここからはアセンブラ定義関数rrotateのチェック用プログラム．

```
#include        <bdscio.h>

main()
{
        int     i;
        unsigned        a;
        a = 1;
        for ( i=0 ; i<16 ; i++ )
        {
                printf ( "rot %d= %d¥t", i, a );
                a = rrotate (a);
        }
}
```

ここまではバッチファイルCASM.SUBによって自動的に行なう事もできる．

```
A>cc test
```
← test.c をコンパイル

```
BD Software C Compiler v1.50a (part I)
  35K elbowroom
BD Software C Compiler v1.50 (part II)
  32K to spare
```

```
A>clink test rrotate ←———— test.crlとrrotate.crl
                               をリンクする.
BD Software C Linker   v1.50
Linkage complete
  43K left over


A>test ←———— test.comを実行する.
              正しい値が求まっている.

rot 0= 1        rot 1= -32768  rot 2= 16384   rot 3= 8192    rot 4= 4096
rot 5= 2048     rot 6= 1024    rot 7= 512     rot 8= 256     rot 9= 128
rot 10= 64      rot 11= 32     rot 12= 16     rot 13= 8      rot 14= 4
rot 15= 2

A>
```

操作例5-1で示したようにまず，RROTATE.CSMというソースファイルをエディターで作成します．CASM.COMにより，これをASMでアセンブルできる形式に変換します．この結果得られたRROTATE.ASMはリスト5-2に示すようになります．

リスト5-2　RROTATE. ASM

```
TPALOC          EQU     0100H


                ORG     TPALOC+200H
                DB      0,0,0,0,0
RROTATE$BEG     EQU     $-TPALOC

                DB      0
                DW      RROTATE$END-$-2
RROTATE$STRT    EQU     $
RROTATE$EF$RROTATE      EQU     RROTATE$STRT

RROTATE$STRTC   EQU     $
        pop     d
        pop     h
        push    h

        mov     a,l
        rar
        mov     a,h
        rar
        mov     h,a
```

```
        mov     a,l
        rar
        mov     l,a

        push    d
        ret

RROTATE$END     EQU     $
                DW      0


END$CRL         EQU     $-TPALOC
SECTORS$ EQU ($-TPALOC)/256+1 ;USE FOR "SAVE" !.

                ORG     TPALOC

; Directory:
                DB      'RROTAT','E'+80H
                DW      RROTATE$BEG
                DB      80H
                DW      END$CRL
                END
```

このプログラムはもともとリロケータブルなので，前後にヘッダーが付いただけになっていますが，JMPやCALLなどのアドレスを含む命令が存在するとその間にラベルを挿入し，リロケータブルな形に変換します．このファイルをASMによってアセンブルし，DDTによってHEXファイルをバイナリ形式に変換します．HEXファイルではアドレスが若干前後するため，細かいエラーチェック機能があるLOADコマンドでは変換できません．

これらの操作はかなり面倒なので，CASM.SUBというサブミットファイルが用意されており，自動的に行なうこともできるようになっていますが，DDTでバイナリ形式に変換した後，ファイルにセーブするためにはどうしてもSAVEコマンドを使わねばならないので，この引数だけは手で与える必要があります．この時，引数の値がわからないと困りますから，わざとアセンブルにエラーがでるようになっています．

```
S0003=   SECTORS$  EQU  ($-TPALOC)/256+1
```

このようなエラーメッセージが必ずコンソール上に出力されます．この最初のS0003が引数の3を示しています．16進数ですから，9より大きい時は10進数に変換しなければなりません．

この処理を図示すると図5-3のようになります．

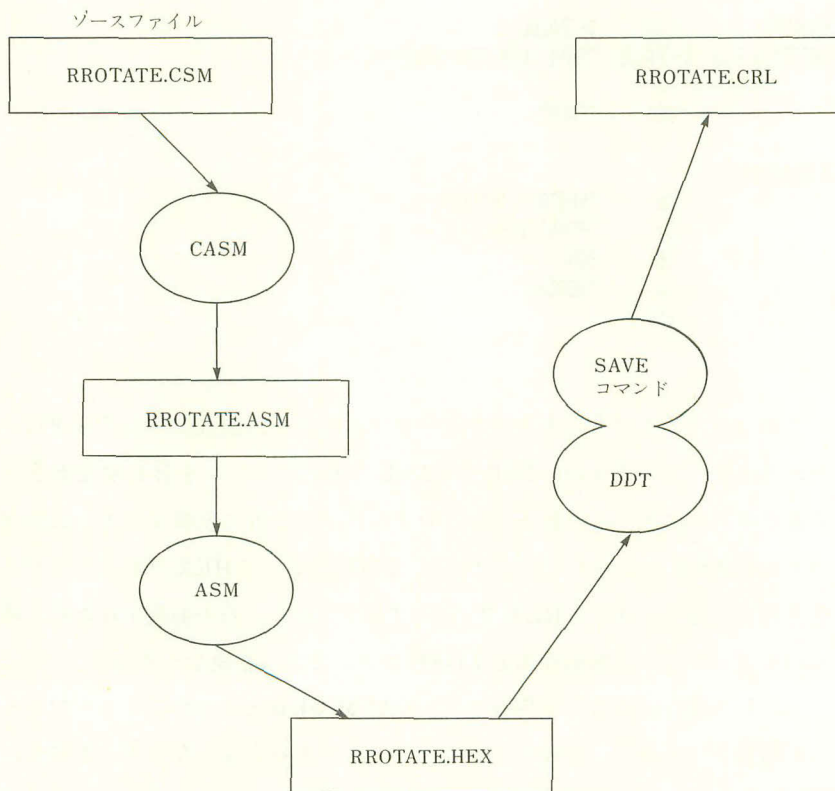



図5-3　CASMの使い方

なお，CASMにはオプションが3つあります．コマンドラインで，これらのオプションを指定する事ができます．

−c　コメントを付加する.

−f　CMAC.LIBを操作する（CASMの古いバージョン用).

−o　出力ファイル名を指定します. −oに続いて, ファイル名を記述します. ただし, 拡張子は不要で必ず.ASM”となります.

5-2 アセンブラプログラムの書き方

CASMを利用する時の書式はほぼ決まっています. 関数名をFUNCTIONで登録し, “ENDFUNC”あるいは“ENDFUNCTION”迄の間にアセンブラのソースプログラムを書きます. もし, 他の（アセンブラ, あるいはCで定義された）関数を呼び出す場合には, “FUNCTION”の直後に利用する関数名を“EXTERNAL”に続けて書かねばなりません.

表5-4にCASMで記述するプログラムの書式を示します.

表5-4　CASMのソーステキストの書式

```
BDOS      EQU  5            定数の宣言はFUNCTION
REBOOT    EQU  0            とENDFUNCの外でする.
      :
FUNCTION  program           関数の宣言
EXTERNAL  function          利用する外部関数の宣言
      :
  〈プログラム本体〉
      :
ENDFUNC                     関数の終了宣言
```

ここで, 呼び出す関数が複数の場合には,

```
EXTERNAL extfunc1,extfunc2,………
```

のように, カンマで区切りながら関数名を列挙するか, EXTERNAL行を続

けて，

```
EXTERNAL    extfunc1
EXTERNAL    extfunc2
```

とします．

プログラムを書く上で注意しなければならないのは，Cプログラムからの引数の受渡しとローカル変数の設定です．BDS Cでは引数やローカル変数はスタック上に設定されますから，その取り出しが厄介です．まず，関数が呼び出された時スタックは図5-5のような状態になっています．

program(a，b)：　〈スタックの状態〉

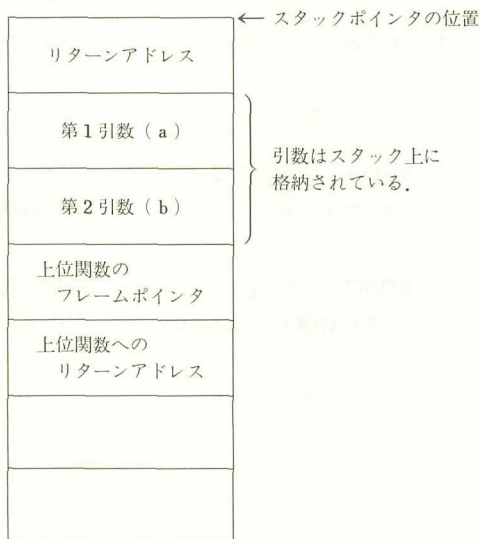



図5-5　呼び出された時のスタックの状態

また，BDS CではBCレジスタをローカル変数の位置を示すフレームポインタとして用いているため，BCレジスタの値を必ず保存しておかねばなり

ません．一般には呼ばれた関数で，最初にPUSH Bを実行しますので，この上にBCレジスタの値がPUSHされています．

ただし，BCレジスタのPUSHは必ず実行しなければならないわけではなく，RROTATE.CSMのようにBCレジスタを用いないアセンブラ定義関数では，なくても構いません．

さて，関数の中でしなければならない処理は一般に次のようなものです．

(1) ローカル変数とフレームポインタの設定

ローカル変数をアセンブラ定義関数の中で利用する場合，スタック上からその領域を確保します．もし，4バイトのローカル変数を確保したいとすれば，

(ザイログニーモニック)	(インテルニーモニック)
PUSH BC	PUSH B
LD HL, −4	LXI H, −4
ADD HL, SP	DAD SP
LD SP, HL	SPHL
LD B, H	MOV B, H
LD C, L	MOV C, L

というプログラムを先頭で実行します．ここで，BCレジスタにはそのスタックポインタの内容がコピーされますから，

BC ⟶ 1バイトめ
BC＋1 ⟶ 2バイトめ
BC＋2 ⟶ 3バイトめ
BC＋3 ⟶ 4バイトめ

のようにローカル変数へのアドレスとなります．

この時，引数は

BC＋8，BC＋9　⟶　第1引数
BC＋10，BC＋11　⟶　第2引数
⋮

で参照されます．なお，注意しなければならないのは引数はすべて2バイトであるという事です．char型の変数でも上位8ビットに00Hが付け加えられ，2バイトに拡張されています．

なお，BC＋4からBC＋7番地までの4バイトが空いていますが，これは

BC＋4，BC＋5　⟶　退避したBCレジスタの内容
BC＋6，BC＋7　⟶　上位関数へのリターン番地

が格納されており，変更してはなりません．

(2)　関数の終了

関数が終了し，上位関数へ戻る時は必ずスタックをもとの位置に戻さなければなりません．4バイト分ローカル変数を使用したとすると，

（ザイログニーモニック）	（インテルニーモニック）
EX　DE, HL	XCHG
LD　HL, 4	LXI　H, 4
ADD　HL, SP	DAD　SP
LD　SP, HL	SPHL
EX　DE, HL	XCHG
POP　BC	POP　B
RET	RET

とします．HLレジスタは関数の戻り値が格納されるレジスタですから，XCHG命令を用いてDEレジスタに内容を移し，一旦退避しています．

勿論，戻り値が不要な時には退避する必要はありません．

(3) Z80 CPUにおけるローカル変数の設定

ローカル変数の処理は8080 CPUではアセンブラでプログラムを作成する場合には大きな負担となります．しかし，Z80CPUを用いたパソコン用のプログラムであれば，インデックスレジスタIXあるいはIYを用いてもう少し楽にできます．プログラム先頭で，

```
PUSH    BC
LD      IX,-4
ADD     IX,SP
LD      SP,IX
```

とすれば，IXがローカル変数へのポインタになりますから，

```
LD      L,(IX+0)
LD      H,(IX+1)
```

あるいは

```
LD      (IX+0),L
LD      (IX+1),H
```

などの命令で簡単にローカル変数や引数をアクセスできます．BCレジスタもプログラム用に使えます．ただし，Z80ではインデックスレジスタを使う命令はバイト数が多いので，見かけ上大きなプログラムとなりますが，プログラムのステップ数は減りますし，何よりプログラムが簡潔になり，アセンブラでは利用しにくいローカル変数が使いやすくなります．また，プログラムの組み方にもよりますが，ローカル変数のアドレスを計算しなくて済む分，実行速度は速くなるでしょう．関数の終了の方法は同じです．

なお，アセンブラ定義関数の中でさらにアセンブラ定義関数を呼び出し，

両者ともIXレジスタを使っているなら，IXレジスタを退避する必要があります．また，IX，IYレジスタのオフセットアドレスは8ビットですから二百数十バイト程度しかローカル変数をアドレッシングできません．それ以上に必要な場合は8080と同じ方法をとるのが良いでしょう．

BDS CはC言語の仕様を満たすため，関数単位でかなり面倒な初期設定を行ないます．またいずれにしろローカル変数は使いにくいので，関数のアセンブラ定義をする場合はむしろリエントラントな構造を諦め，8080，Z80の得意な，絶対番地をアクセスするプログラミングをした方が良いでしょう．

(4)　引数の取り出し

BCレジスタ，IXレジスタなどにスタックポインタをコピーし，そのオフセットを計算して引数を取り出す方法はあらゆる場合に対応できますが，余り多くの引数を必要としないアセンブラ定義関数を作る場合も数多くあります．

引数が1つしかない場合には，BCレジスタ退避の前に

〈ザイログ〉	〈インテル〉	
POP　DE	POP　D	；リターンアドレス
POP　HL	POP　H	；引数
PUSH HL	PUSH H	
PUSH DE	PUSH D	

とする事で簡単に引数をHLレジスタに呼び出せます．8080の命令だけでは1つしか取り出す事ができませんが，Z80であれば，裏レジスタやIX,IYレジスタもありますから，2～3個取り出せます（余り多くレジスタに取り出してしまうと使えるレジスタの数が減るのでかえってプログラミングが面倒になります）．

(5) 他の関数の呼び出し

まず，参照する関数名は定義の先頭で，“EXTERNAL”指定をしておく事が前提で，引数は逆順にスタックにプッシュします．例えば，

```
extfunc (a,b,c);
```

であれば，c の値，b の値，a の値の順に PUSH し，extfunc を CALL します．リターン時には HL レジスタに戻り値が入っていますから，POP D などを PUSH した引数の数だけ行なって，スタックを元の状態に戻し次のプログラムに続けます．

引数の設定やスタックの回復を誤ると暴走しますので十分に注意してください．これらはプログラマーの責任になります．

操作例5-6に printf 関数を利用する例を示します．これを参考にコツを掴んでください．

操作例 5-6 EXTERNAL の使い方

```
A>type csmprint.csm

        FUNCTION        main
        EXTERNAL        printf

        push    b               ;frame pointer push
        mvi     c,8             ;counter 8
        mvi     b,1             ;test data 1

test1:
        mov     l,b             ;test data push
        mvi     h,0
        push    h  <──────────────────── 第3引数
        mov     l,c             ;counter push
        push    h  <──────────────────── 第2引数
        lxi     h,message       ;chars push
        push    h  <──────────────────── 第1引数
        call    printf <──────────────── BCレジスタは必らず保存されるので
                                         退避する必要はない．
        pop     d               ;stack return ┐
        pop     d                             │ 引数3つ分の
        pop     d                             ┘ スタック回復．
```

```
        mov     a,b             ;b=b*2
        add     a
        mov     b,a

        dcr     c               ;if c!=0 then test1
        mov     a,c
        ora     a
        jnz     test1

        pop     b               ;frame pointer pop
        ret

message:                        ;test message
        db      'Count %d: b= %d',10,0

        ENDFUNCTION
```

第3引数
第2引数
第1引数は文字列の頭のアドレス

```
A>submit casm csmprint

A>XSUB

A>CASM CSMPRINT
BD Software CRL-format ASM Preprocessor v1.50
Processing the MAIN function...
CSMPRINT.ASM is ready to be assembled.

(xsub active)
A>ASM CSMPRINT.AAZ

CP/M ASSEMBLER - VER 2.0
S0003 =         SECTORS$ EQU ($-TPALOC)/256+1 ;USE FOR "SAVE" !.
0109
001H USE FACTOR
END OF ASSEMBLY

(xsub active)
A>DDT CSMPRINT.HEX

DDT VERS 2.2
NEXT  PC
0350 0000
-G0
```

```
(xsub active)
A>ERA CSMPRINT.ASM
A>ERA CSMPRINT.HEX

A>save 3 csmprint.crl

A>clink csmprint

BD Software C Linker   v1.50
Linkage complete
  43K left over

A>csmprint

Count 8: b= 1
Count 7: b= 2
Count 6: b= 4
Count 5: b= 8
Count 4: b= 16
Count 3: b= 32
Count 2: b= 64
Count 1: b= 128

A>
```

正しく内容が出力される.

5-3 アセンブラの利用

一般にC言語でアセンブラを利用するのは,

(1) 特殊な外部機能を制御する場合.

(2) 特にプログラムの高速化をはかりたい時.

(3) アセンブラで記述した方が簡単な場合.

などが主になります. しかし, BDS Cでは次のような用途もあります.

(4) BDS Cで削除されている機能のバックアップ

(5) アセンブラプログラムでBDS Cライブラリ（標準関数）だけを利用する場合.

これらは，本来の使用法ではありませんが，BDS Cでは初期値の設定ができないため非常に無駄な処理をしなければならない場合があります．このような時，メモリと実行時間の無駄を省けます．

例えば，余り精密でなくても良いから高速なsin関数が欲しい時があります．このような場合に最も簡単なのはあらかじめsin関数テーブルを用意しておく方法ですが，BDS Cでは変数に初期値を入れられないため，initw関数などを使いわざわざ数値を文字列（アスキーコード）に変換してしまうため，3倍のテーブルメモリとプログラムを必要とし，さらに初期設定のために実行時間がかかります．

このような内容のデータは変更する性格のものではありませんから，アセンブラで定義し，じかにプログラム中に入れてしまう方が得策です．

リスト5-7にテーブルサーチのサンプルを示します．0から7までの数値をデコードするプログラムです．一般的にはループで行ないますが，テーブルを作っておく方が高速です．このプログラムでは，

引数		出力
0	−	01H
1	−	02H
2	−	04H
3	−	08H
4	−	10H
5	−	20H
6	−	40H
7	−	80H

となります．

リスト5-7 アセンブラによるテーブルサーチ

```
        FUNCTION        DECODE

        POP     H
        POP     D       ;arg
        PUSH    D
        PUSH    H

        LXI     H,TABLE
        DAD     D
        MOV     L,M
        MVI     H,0
        RET

TABLE:  DB      01H,02H,04H,08H
        DB      10H,20H,40H,80H

        ENDFUNC
```

なお，初期値設定の問題について文字列だけは，

```
a="This is string.";
```

とすると，文字列が関数の最後にコードで付け加えられ，aに文字列の先頭アドレスが入ります．5-4節のリスト5-9（ZCASM）にこの特徴をうまく使った部分がありますので，最後に定義しているinitdim関数を参照してください．

次に(5)は，プログラムサイズが厳密に決まっている場合や，特にプログラムの高速化を狙う時に行ないます．頻繁にあるとは言えませんが，元々アセンブラ関数が非常に多い時や，アセンブラのライブラリを利用したい時にこういう形式になる場合があります．前節の操作例5-6はその例です．

5-4 CASMの問題点

ザイログ社のZ80ニーモニックになじんだユーザーにとって，BDS Cのア

センブラはかなり面倒なものです．慣れたニーモニックが使えない上，Z80の機能を使いたい場合にはハンドアセンブルする必要があります．

前項で述べたように，Z80はインデックスレジスタによりすっきりとローカル変数を扱う事ができますから，非常に便利です．私自身,BDS Cを使いはじめたきっかけはアセンブラの代替という目的からでしたから，それまで作成していたザイログニーモニックのソースプログラムが一切使えないというのはかなり痛手で，導入の直接のきっかけとなったソフト開発には使えませんでした．

CASMはソースで供給されていますので，Z80用に改造する試みはZ80のユーザーなら誰しも考える所ですが，CASMは最初からアルゴリズム自体がインテルニーモニック用に考えられているため，Z80オリジナル命令への対処ができず，同一の方法では困難です．

また，CASMの使い勝手もかなり悪く，とりわけSAVEコマンドを手でタイプインしなければならないのは誤りを誘いやすく，デバッグ時に何度も行なう際には非常に危険です．また，プログラム開発はコンパイル，リンクの一連のシーケンスをSUBMITファイルにしておく事が多いものですが，アセンブラのソースファイルがあるたびに中断されます．これはLOADコマンドが使えず，DDTでHEXファイルをCRLファイルに変換するという仕様自体の欠点です．

さらに，小さなバグもいくつかあり，注意する必要があります．

(1)　“FUNCTION”は行の第1文字めから書き始めてはいけない．

(2)　ラベルは必ず行の第1文字めから書き始めなければならない．

(3)　DWのあとに，ラベルは1つしか書けない．

(4)　オペランドの式評価において，正しくリロケータブル変換されない場

合がある.

```
LABEL1:  ……
LABEL2:  ……
                LXI  H,LABEL2-LABEL1
```

というプログラムは良く用いられます. HLレジスタにLABEL2とLABEL1の間にあるバイト数を格納したいのですが,

```
LXI  H,FUNC$LABEL2-LABEL1
```

と変換されてしまいます. これはバイト数ですから, リローケーションの必要は無いわけですが, 残念ながらリローケーションパラメータもセットされてしまいます.

これは本質的な誤りですし, 第一, アセンブルでエラーがでます (未定義ラベルエラー). このような記述が必要な場合はバイト数を数えて, ダイレクトに数値で書かねばなりません.

(5) EQU宣言のオペランドはリロケーションの対象にならない.

同様なプログラムで,

```
BYTES  EQU  LABEL2-LABEL1
             LXI  H,BYTES
```

としても, EQU以降については全くラベル名変換の処理が行なわれないため, 使えません (ASMでエラーとなります).

(6) 16ビット値をバイト変換して使えない.

アブソリュートアセンブラでは

```
LABEL:  ……
```

```
MVI  A,LABEL AND OFFH
MVI  B,LABEL SHR 8
```

などと日常的に用いられますが，これらは使用できません．これは，リロケータブルアセンブラなどでも同じですから，さほどの問題とはなりませんが，注意する必要はあります．

(7)　外部変数が使いにくい．

以上のように，CASMは主にラベルの扱いについて若干問題があります．大抵アセンブル時にエラーとなってはくれますが，SUBMITで実行した時は画面にエラーメッセージが出るかどうか監視していなければなりません（エラーメッセージ自体がスクロールして消えてしまいます）．

5-5　Z80アセンブラとのリンク

前に述べたように，CASMではZ80アセンブラ（ザイログニーモニック，インテル拡張ニーモニック）とは，リンクする事ができません．例えばFUNCという関数の場合，LXI命令があると次のように変換されます．

```
           LXI   H,ADDRESS
            :
ADDRESS:   DB    O
```

というソースプログラムは，

```
FUNC$ROOO  EQU   $+1-FUNC$STRT
           LXI   H,FUNC$L$ADDRESS
            :
FUNC$L$ADDRESS  EQU  $-FUNC$STRT
           DB    O
```

のように変換されます．ここで，FUNC＄STRTというのは，この関数モジュールにおけるプログラムの先頭アドレスで，すべてラベルをFUNC$STRTからの距離に置き換え，リロケータブルにしています．さらに，LXI命令の直前にFUNC＄ROOOを定義する1行が追加されており，このFUNC＄ROOOがあとでリンクする際に変更しなければならないリロケーションアドレスとして，モジュールの最後にあるリロケーションリストの中に登録されます．

さてこの時，HLレジスターではなく，IXレジスターだとどうなるでしょう．おわかりだと思いますが，この場合は4バイト命令で，後半の2バイトがリロケーションの必要なデータになりますから，FUNC＄ROOOのリロケーションアドレスは＄＋1－FUNC＄STRTではなく，＄＋2－FUNC＄STRTでなければなりません．

CASMのアルゴリズムでは，4バイト命令の場合の対処が非常に複雑となり，Z80対応の拡張インテルニーモニックならばともかく，ニーモニックとオペランドをすべて解析する必要がでてきます．これでは，開発パワーがかなりかかります．

私自身この問題で，長い間悩んできましたが，Z80用CASMを作成する適当な方法を思い付きましたので，今回を機会にZ80用のCASMプリプロセッサを作成しました．特にα-CのユーザーはCASMそのものを持っていませんから，利用価値が高いと思います．

さて，Z80用のCASMをZCASMと名付ける事にし，プログラムの考え方を説明します．

(1)　リロケーションの方法

8080では，2バイトのイミーディエイトデータ(数置)を持つ命令は必ず3バイトでしたが，Z80では3バイトと4バイトのものがあります．この時，イミーディエイトデータは必ず最後の2バイトである事を利用し，リロケートします．

(2)　2パスプリプロセッサ

CASMでは，高速に処理するために1パス方式を採用していますが，そのかわりDDTでなければHEXファイルのロードができないという欠点がありました．そこで，ZCASMでは，やや処理時間が遅くなりますが，2パス方式を採用しLOADコマンドによってファイル形式の変換が行なえるようにしました．特に，アセンブラとして，MACRO-80(とLINK-80)のようにダイレクトにバイナリーファイルを出力できる機能を持っているものを使う場合には，DDTやLOADコマンドを使う必要もありません．

(3)　ラベル参照方法の改善

ラベルからラベルまでのバイト数を参照したい場合など，アドレスを内容とするラベルを用いて，イミーディエイトデータを表現する事があります．このような場合はアドレスと解釈されてリロケートされては困ります．そこで，次のような記述が可能になっています．

```
LABEL1: .....
LABEL2:
BYTES   EQU   LABEL2-LABEL1
        LD    HL,BYTES
```

この方法では，LABEL 1，LABEL 2がリロケートされますが，EQUで定義されたBYTESの値は変化しませんので，正しく変換されます．なお，

```
LD        HL,LABEL2-LABEL1
```

と記述すると，ラベルが2つともリロケートされますので一見正しいように見えますが，残念ながらリロケーションリストに登録されてしまいますので正しく動作しません．

CASMと異なり，EQU擬似命令，一般の命令いずれのオペランドに対してもすべてアドレスを示すラベルはリロケートされます．

(4) ラベル認識アルゴリズム

ZCASMでは，アドレスを示すラベルであるかのどうかを決定するのに，次のようなアルゴリズムを用いています．

PASS 1…ソースファイルを検索し，1行の最初の単語がニーモニックであるかどうかを調べます．もし，ニーモニックでなく，かつ2つめの単語がEQUでなければこれをアドレスを示すラベルと判定し，メモリ中のラベルリストに加えます．この操作は関数ごとに行います．

PASS 2…ソースファイルを再び頭から読み出し，出力ファイルを作成する作業を行います．1行の2番め以降の単語がラベルリストにあれば，これをリロケートラベルに変換し，この次の行に無条件にリロケートリストに入れるラベルを挿入します．

(5) 使用方法

基本的な使用方法はCASMと全く同じで，プログラムの書式も同じです．

```
A>ZCASM filename
```

で，ファイルを変換します．この時，CASMと同じように入力ファイルは拡張子を．“. CSM”とします．ただし，filenameには”.CSM” は記述しません．出力ファイル名は標準では“.ASZ”となります．これは私がいつも利用しているZ80アブソリュートアセンブラMR−ASMの入力ファイル名ですが，MACRO−80を使う方のため，−mというオプションを用意しています．

```
A>ZCASM filename -m
```

とすると，filename. MACというファイルが出力され，内容もMACRO−80用に少し変更されたものになります．オプションは−m以外ありません．

操作例5-8にZCASMの使い方を示します．中で使われているZCASM. SUB, ZCMR. SUBは一連のシーケンスをまとめたバッチファイルで，これを使うと便利です．

操作例5－8　ZCASM操作例

```
A>TYPE PRINT.CSM

        FUNCTION        MAIN
        EXTERNAL        PRINTF

        LD      IX,WORD
        PUSH    IX
        CALL    PRINTF
        POP     DE
        RET

WORD:   DB      'THIS IS ZCASM TEST',0

        ENDFUNC
```

Z80ニーモニックのサンプルプログラム．

```
A>ZCASM PRINT

ARMAT Z80-CRL preprocessor v1.0
Pass 1: End
Pass 2:
 -Processing the MAIN function..
  PRINT.ASZ is ready to be assembled.


A>MRASM PRINT
Armat Z80 Assembler - VER 1.1
  (c) Armat co. 1985 All Rights Reserved.

No Fatal error(s)
```

MRASMはアーマット社で開発されたASMコンパチブルのZ80用アセンブラ．

```
A>LOAD PRINT
FIRST ADDRESS 0100
LAST  ADDRESS 0339
BYTES READ    0043
RECORDS WRITTEN 05


A>REN PRINT.CRL=PRINT.COM

A>CLINK PRINT
BD Software C Linker   v1.50
Linkage complete
  43K left over
```

```
A>PRINT

THIS IS ZCASM TEST


A>TYPE ZCMR.SUB
```

```
ZCASM $1
MRASM $1.AAZ ;CHANGE IF YOU USE OTHER DISK
LOAD $1
ERA $1.CRL
REN $1.CRL=$1.COM
;$1.CRL IS READY
```

MRASMを使う場合は，このサブミットファイルを使えば非常に便利である．

```
A>TYPE ZCASM.SUB
```

```
ZCASM $1 -M
M80 =$1
ERA $1.MAC
L80 /P:100,$1,$1.CRL/E/N
;$1.CRL IS READY
```

M80(マクロ80)を使う場合は，このサブミットファイルを用いる．

なお，ZCASMでは，プログラムの頭にある

```
#define Z80 1
```

を0に書き直してコンパイルする事で，ASM用のインテルニーモニック版，すなわちCASMと殆ど同じものになります．これは，実はデバッグ用で，CASM用に書かれたプログラムをこの8080版ZCASMにかけ，私の考えたアルゴリズムでCASMと全く同じオブジェクトが発生する事を確認する目的で追加したものです．ちなみに，BDS Cのアセンブラ定義関数のソースリスト(DEFF2A-DEFF2D)をテストした所，DEFF2A.CSMだけは同一になりません．これは，DEFF2A.CSM側の問題で，CMPHDというBDS.LIBで定義されているサブルーチンと殆ど同じ機能を持つサブルーチンをうっかり再定義しているためです．このどちらかをCALLするかがCASMとZCASMで異なるのです．CASMではBDS. LIB側，ZCASMではその関数内定義ラベ

ルをCALLします．動作は全く同じですが，私は重複したラベルを定義した時には，その関数内で定義されているローカルラベルを優先する方が自然ですし，誤りも少なくなると思います．

また，DEFF 2 Dは長過ぎるラベルを使っています．ZCASMで使用できるラベルの長さは，最長8文字ですので，変更してからアセンブルします．なお，ラベルの長さは実際にはアセンブラの機能に左右されます．MACRO-80は長いラベルが使えないので，なるべく7文字以下にしてください．

なお，8080に対応する機能，あるいはZ80に対応する機能がいらない方は，その部分を打ち込む必要はありません．リストをよく見て削除してください．

リスト5-9　ZCASM.C

```
 1: /*********************************************************************
 2:
 3:                    -----  ZCASM.C  -----
 4:
 5:          Z80-ASM to BDS-CRL format translate program
 6:
 7:          Author:         Tsu.Mitarai     01/sep/1985
 8:          Arranged:                       16/jan/1986
 9:
10:                  A>cc zcasm.c -e3800
11:                  A>clink zcasm
12:                          (or A>l2 zcasm)
13:
14:          < Caution - The difference from CASM >
15:
16:          1:      Same symbol name
17:                  ZCASM - local symbol is prior.
18:                  CASM  - global symbol is prior.
19:          2:      Label length
20:                  ZCASM - 8 chars (desirably, 7 chars)
21:
22: *********************************************************************/
23:
24: #include        <bdscio.h>
25:
26: #define Z80     1               /* if 8080 then 0               */
27:
28: #if Z80
```

```
29:
30: #define JPMNE   "JP"            /* Z80 absolute jump mnemonic   */
31: #define NEM2    16
32: #define NEM3    40
33: #define NEM4    23
34: #define SUBO    8
35:
36: #else
37:
38: #define JPMNE   "JMP"           /* 8080 absolute jump mnemonic  */
39: #define NEM     100
40: #define NEM2    18
41: #define NEM3    57
42: #define NEM4    10
43: #define SUBO    8
44:
45: #endif
46:
47: #define TITLE   "Armat Z80-CRL preprocessor v1.0¥n"
48:
49: #define INEXT   ".CSM"          /* input file extention         */
50:
51: #if Z80
52:  #define        OUTEXT1 ".ASZ"  /* for Armat Z80 Assembler      */
53: #else
54:  #define        OUTEXT1 ".ASM"
55: #endif
56:
57: #define OUTEXT2 ".MAC"          /* for Macro-80                 */
58:
59: #define FNCSIZ  64              /* Max 64 functions             */
60: #define FNCLEN  8               /* fnc name length              */
61: #define LABSIZ  500             /* Max 500 labels               */
62: #define LABLEN  8               /* Label length                 */
63: #define EXTSIZ  200             /* Max 200 external functions   */
64: #define EXTLEN  8               /* external label length        */
65: #define TOKLEN  128             /* Max token length             */
66: #define QUOTE   0x27            /* quotation '                  */
67: #define DQUOTE  0x22            /* double quotation             */
68:
69: #define TRUE    1
70: #define FALSE   0
71: #define FUNC    1
72: #define EFUNC   2
73: #define EXTN    3
```

```
 74: #define INCL    4
 75: #define PEND    5
 76:
 77: char    inname [ 20 ];          /* input file name           */
 78: char    outname [ 20 ];         /* output file name          */
 79: char    incname [ 20 ];         /* include file name         */
 80: char    lbuf [ 256 ];           /* line input buffer         */
 81:
 82: char    word [ TOKLEN + 1 ];    /* token word                */
 83: char    *tokenpnt;              /* pointer to token          */
 84: char    *ltokenpnt;             /* last pointer to token     */
 85: char    prosflag;               /* process flag              */
 86: char    rflag;                  /* relocate flag             */
 87: char    eflag;                  /* EQU flag                  */
 88: char    jrflag;                 /* JR or DJNZ flag           */
 89:
 90: char    mflag;                  /* using Macro-80 flag       */
 91:
 92: char    fncname [ FNCLEN + 1 ]; /* function name             */
 93:
 94: unsigned        fncnt;          /* function counter          */
 95: unsigned        nlabel;         /* next label number         */
 96: unsigned        nextn;          /* next external             */
 97: unsigned        extcnt;         /* external counter          */
 98: int     relcnt;                 /* relocation counter        */
 99: char    incflag;                /* file include flag         */
100:
101:
102: struct  fntable
103:         {
104:                 unsigned        label;  /* ltable[n] n       */
105:                 unsigned        extn;   /* etable[n] n       */
106:         }
107:         fnct [ FNCSIZ ];                /* struct func table */
108:
109: char    ltable [ LABSIZ ][ LABLEN + 1 ];        /* labels       */
110: char    etable [ EXTSIZ ][ EXTLEN + 1 ];        /* externals    */
111: char    ftable [ FNCSIZ ][ FNCLEN + 1 ];        /* func names   */
112:
113: /* mnemonic table        */
114:
115: char    *nem2 [NEM2];
116: char    *nem3 [NEM3];
117: char    *nem4 [NEM4];
118: char    *subo [SUBO];
```

```
119:
120:
121: FILE    ofpin;                        /* input file ( original file ) */
122: FILE    ifpin;                        /* input file ( include file )  */
123: FILE    fpout;                        /* output file                  */
124: FILE    *fpin;                        /* pointer to FILE              */
125:
126:
127: /*      MESSAGES        */
128:
129: #define MACERR  " -Sorry,MACRO is not supported."
130: #define STRERR1 " -String too long in %s."
131: #define STRERR2 " -Missing quote in %s."
132: #define EXTERR  " -EXTERNAL name %s is too long."
133: #define EXTERR2 " -Too many externals."
134: #define LABERR  " -LABEL %s is too long."
135: #define LABERR2 " -Too many labels."
136: #define INCERR1 " -Only one level of inclusion is supported."
137: #define INCERR2 " -Missing include file %s."
138: #define DBLERR  " -Missing ENDFUNC in %s."
139: #define FNCERR  " -Too long name ¥"%s¥"."
140: #define FNCERR2 " -Too many functions."
141: #define OPENERR " -Can't open %s."
142: #define CRETERR " -Can't creat %s."
143:
144:
145: main(argc,argv)
146: int     argc;
147: char    *argv[];
148: {
149:         printf (TITLE);
150:         initdim ();
151:
152:         if (argc != 2 && argc != 3)
153:         {
154:                 printf ("Usage: ZCASM filename [-m]¥n");
155:                 printf ("¥t-m: for Macro-80¥n");
156:                 kexit();
157:         }
158:
159:         if (argc == 3 && !strcmp (argv[2],"-M"))
160:                 mflag = TRUE;
161:         else
162:                 mflag = FALSE;
163:
```

```
164:         strcpy (inname, argv[1]);
165:         strcpy (outname,argv[1]);
166:
167:         strcat (inname, INEXT);                              /* input file */
168:
169:         if (mflag)      strcat (outname,OUTEXT2);
170:         else            strcat (outname,OUTEXT1);           /* output file */
171:
172:         if (ERROR == fopen (inname, ofpin))
173:         {
174:                 printf (OPENERR, inname);
175:                 kexit();
176:                                 /* file open error       */
177:         }
178:
179:         if ( ERROR == fcreat (outname, fpout))
180:         {
181:                 printf (CRETERR, outname );
182:                 kexit();
183:                                 /* disk full ?           */
184:         }
185:
186:         printf ("Pass 1:");
187:         pass1 ();                   /* search function name */
188:         printf (" End¥n");
189:
190:         fclose (ofpin);
191:         fopen (inname, ofpin);
192:
193:         printf ("Pass 2:¥n");
194:         pass2 ();                   /* main routine          */
195:         printf ("  %s is ready to be assembled.¥n",outname);
196:
197:         fclose (ofpin);
198:         fclose (fpout);
199: }
200:
201:
202: /*-----------------------------
203:         Kill submit file & exit
204: -----------------------------*/
205:
206: kexit()
207: {
208:         unlink ("$$$.SUB");
```

```
209:         exit();
210: }
211:
212: /*-----------------------------------------
213:         GET TOKEN FUNCTION
214:         in:     word (token buffer address)
215:                 tokenpnt (G.input line pointer)
216:         out:    TRUE (if ok)
217:                 FALSE (if line end or comment)
218:                 word (token word)
219:                 ltokenpnt (G.last pointer)
220:                 tokenpnt (G.next pointer)
221: -----------------------------------------*/
222:
223: gettoken(word)
224: char    *word;
225: {
226:         ltokenpnt = tokenpnt;
227:         skipsp ();
228:
229:         if ( *tokenpnt == 0 || *tokenpnt == ';')
230:         {
231:                 return (FALSE);
232:         }
233:         else if ( *tokenpnt == QUOTE || *tokenpnt == DQUOTE)
234:                 chtoken (word);
235:
236:         else
237:         {
238:                 do      *word++ = toupper (*tokenpnt);
239:                 while ( ischar (*tokenpnt++) && ischar (*tokenpnt));
240:                 *word = 0;
241:         }
242:         return (TRUE);
243: }
244:
245:
246: /*-----------------------------------------
247:         GET TOKEN - When token is string
248: -----------------------------------------*/
249:
250: chtoken(word)
251: char    *word;
252: {
253:         int     i;
```

```
254:            char    ch;
255:            i = 0;
256:
257:            ch = *word++ = *tokenpnt++;
258:            do
259:            {
260:                    *word++ = *tokenpnt;
261:                    if ( i++ > TOKLEN )
262:                    {
263:                            printf (STRERR1,fncname);
264:                            kexit ();
265:                    }
266:                    if ( ch == 0 )
267:                    {
268:                            printf (STRERR2,fncname);
269:                            kexit ();
270:                    }
271:            }
272:            while ( ch != *tokenpnt++ );
273:            *word = 0;
274: }
275:
276:
277:
278: /*--------------------------------------------
279:            IS IT CHAR?
280:            in:     ch
281:            func:   if ch is A-z,or 0-9 return TRUE
282:                    else FALSE
283: --------------------------------------------*/
284:
285: ischar(ch)
286: char    ch;
287: {
288:            if (isdigit (ch))                return (TRUE);
289:            if (isalpha (ch))                return (TRUE);
290:            if ( ch == '.' || ch == '@')     return (TRUE);
291:            if ( ch == '$' || ch == '_')     return (TRUE);
292:            if ( ch == ''' || ch == '"')     return (TRUE);
293:            return (FALSE);
294: }
295:
296:
297: /*---------------------
298:            Skip space
```

```
299: -----------------------*/
300:
301: skipsp()
302: {
303:         while ( *tokenpnt == ' '  ||
304:                 *tokenpnt == '¥t' ||
305:                 *tokenpnt == ':'     )
306:
307:                 tokenpnt++;
308: }
309:
310:
311: /*------------------------------
312:         Pass 1
313:         func:   regist externals
314:                 search labels
315: --------------------------------*/
316:
317: pass1()
318: {
319:         char    *adr;
320:         char    toknum;
321:
322:         fpin = ofpin;
323:         fncnt = 0;              /* function table counter initial */
324:         prosflag = FALSE;
325:
326:         while (nextget ())
327:         {
328:                 word[0] = 0;
329:                 tokenpnt = lbuf;
330:
331:                 if (gettoken (word))
332:                 {                                       /* first token  */
333:                         if (PEND == (toknum = cktoken (word)))
334:                                 break;
335:
336:                         if (    prosflag == TRUE ||
337:                                 toknum == FUNC ||
338:                                 toknum == INCL  )
339:                                 parsel(toknum);
340:                 }
341:
342:                 if (gettoken (word))
343:                 {                                       /* second token */
```

```
344:                            if (!strcmp (word,"END"))
345:                                    break;
346:                            if (!strcmp (word,"MACRO"))
347:                            {
348:                                    printf (MACERR);
349:                                    kexit();
350:                            }
351:                    }
352:            }
353: }
354:
355:
356: /*---   Get next line
357:         when include file finish,
358:          return to original file   ---*/
359:
360: nextget()
361: {
362:         char    ff;
363:         ff = fgetl (lbuf,*fpin);
364:         if ( !ff && incflag )
365:         {
366:                 close ( ifpin );        /* include file end     */
367:                 fpin = ofpin;
368:                 incflag = FALSE;
369:                 ff = fgetl (lbuf,*fpin);
370:         }
371:         if ( !ff )      return (FALSE);         /* file end     */
372:         return (TRUE);
373: }
374:
375:
376: fgetl(buf,fp)
377: char    *buf;
378: FILE    *fp;
379: {
380:         char    c;
381:         char    *pnt;
382:         c = fgets (buf,*fp);
383:         pnt = buf + strlen (buf) - 1;
384:         if ( *pnt == '¥n' ) *pnt = 0;
385:         return (c);
386: }
387:
388:
```

```
389: /*     Check key word  */
390:
391: char   cktoken(word)
392: char   *word;
393: {
394:        if (!strcmp ( word, "FUNCTION"))      return (FUNC);
395:        if (!strcmp ( word, "ENDFUNC"))       return (EFUNC);
396:        if (!strcmp ( word, "ENDFUNCTION"))   return (EFUNC);
397:        if (!strcmp ( word, "EXTERNAL"))      return (EXTN);
398:        if (!strcmp ( word, "EXTERN"))        return (EXTN);
399:        if (!strcmp ( word, "INCLUDE"))       return (INCL);
400:        if (!strcmp ( word, "END"))           return (PEND);
401:
402:        return (FALSE);
403: }
404:
405:
406: /*---   pass1 main routine   ---*/
407:
408: parse1(code)
409: char   code;
410: {
411:        switch ( code )
412:        {
413:                case FUNC:
414:                {
415:                        if ( fncnt > FNCSIZ )
416:                        {
417:                                printf ( FNCERR2 );
418:                                kexit();
419:                        }
420:
421:                        gettoken (word);
422:                        if ( strlen (word) > FNCLEN )
423:                        {
424:                                printf ( FNCERR ,word );
425:                                kexit ();
426:                        }
427:
428:                        strcpy ( ftable[fncnt], word);
429:                        fnct[fncnt].label = nlabel;
430:                        fnct[fncnt].extn = nextn;
431:                        prosflag = TRUE;
432:                        break;
433:                }
```

9

10

```
434:
435:            case EXTN:
436:            {
437:                    do
438:                    {
439:                            gettoken (word);
440:                            if (strlen (word) > EXTLEN)
441:                            {
442:                                    printf (EXTERR,word);
443:                                    kexit();
444:                            }
445:                            strcpy ( etable [nextn++], word);
446:                            if (nextn > EXTSIZ)
447:                            {
448:                                    printf (EXTERR2);
449:                                    kexit();
450:                            }
451:                            strcpy ( ltable [nlabel++], word);
452:                            if ( nlabel > LABSIZ )
453:                            {
454:                                    printf (LABERR2);
455:                                    kexit ();
456:                            }
457:                            gettoken (word);
458:                    }
459:                    while ( *word == ',' );
460:                    break;
461:            }
462:
463:            case EFUNC:
464:            {
465:                    ltable [nlabel++][0] = 0;
466:                    etable [nextn++][0] = 0;
467:                    fncnt++;
468:                    prosflag = FALSE;
469:                    break;
470:            }
471:
472:            case INCL:
473:            {
474:                    incset ();
475:                    break;
476:            }
477:
478:            default:
```

```
479:                    {
480:                            if (!ismnem (word))
481:                            {
482:                                    if (strlen (word) > LABLEN)
483:                                    {
484:                                            printf (LABERR,word);
485:                                            Kexit();
486:                                    }
487:                                    strcpy ( ltable [nlabel], word);
488:                                    gettoken (word);
489:                                    if (strcmp (word,"EQU"))
490:                                    {
491:                                            nlabel++;
492:                                    }
493:                            }
494:                    }
495:            }
496: }
497:
498:
499:
500: /*---------------------------
501:            Pass 2
502:            write output file
503: ---------------------------*/
504:
505: pass2()
506: {
507:            int     extcnt;
508:            char    type;
509:            int     i;
510:            char    ff;
511:            char    toknum;
512:
513:            dirout();               /* directory output     */
514:
515:            fncnt = 0;              /* function table counter initial */
516:            extcnt = 0;
517:            fpin = ofpin;
518:            incflag = FALSE;
519:            prosflag = FALSE;
520:
521:            while ( nextget ())
522:            {
523:                    tokenpnt = lbuf;
```

```
524:                    if (gettoken (word))
525:                    {
526:                            if (PEND == (toknum = cktoken (word)))
527:                                    break;
528:
529:                            if (    prosflag == TRUE ||
530:                                    toknum == FUNC ||
531:                                    toknum == INCL   )
532:
533:                                    parse2 (toknum);
534:
535:                            else    fprintf (fpout,"%s¥n",lbuf);
536:                    }
537:                    else    fprintf (fpout,"%s¥n",lbuf);
538:            }
539:            fprintf (fpout, "QED$CRL:¥n");
540:            fprintf(fpout,"END¥n%c",CPMEOF);
541: }
542:
543:
544: /*     Directory output        */
545:
546: dirout()
547: {
548:            unsigned        i;
549:            char    fword[20];
550:            char    lword[2];
551:
552:            fprintf (fpout, ";¥t%s¥n",TITLE);
553:
554: #if Z80
555:            if (mflag)      fprintf (fpout, ".Z80¥n");
556: #endif
557:
558:            if (!mflag)     fprintf(fpout, "ORG¥t100H¥n");
559:                    fprintf (fpout, "TPALOC:¥n");
560:            fprintf (fpout, ";¥tDirectory table¥n");
561:
562:            for ( i = 0; i < fncnt; i++)
563:            {
564:                    first ( ftable[i], fword );
565:                    last ( ftable[i], lword);
566:                    fprintf (fpout, "¥tDB¥t'%s','%s'+80H¥n",fword,lword);
567:                    fprintf (fpout, "¥tDW¥t%s$QBG-TPALOC¥n",ftable[i]);
568:            }
```

```
569:
570:            fprintf (fpout,"¥tDB¥t80H¥n");
571:            fprintf (fpout,"¥tDW¥tQED$CRL-TPALOC¥n¥n");
572: }
573:
574:
575: /*         Take first character      */
576:
577: first(string, word)
578: char       *string;
579: char       *word;
580: {
581:            while ( *word++ = *string++);
582:            --word;
583:            *--word = 0;
584: }
585:
586: /*         Take string without first character      */
587:
588: last(string, word)
589: char       *string;
590: char       *word;
591: {
592:            while ( *string++ );
593:            --string;
594:            strcpy (word,--string);
595: }
596:
597: butlast(string,word)
598: char       *string;
599: char       *word;
600: {
601:            while ( *string )
602:                    *word++ = *string++;
603:            *--word = 0;
604: }
605:
606:
607: /*         Pass2 main routine       */
608:
609: parse2(code)
610: char       code;
611: {
612:            int      i;
613:            switch ( code )
```

```
614:            {
615:                    case FUNC:
616:                    {
617:                            if (prosflag)
618:                            {
619:                                    printf (DBLERR,fncname);
620:                                    kexit();
621:                            }
622:
623:                            else if ( fncnt == 0 )
624:                            {
625:                                    fprintf (fpout,"ORG¥tTPALOC+200H¥n");
626:                                    fprintf (fpout,"¥tDB¥t0,0,0,0,0¥n¥n");
627:                            }
628:                            relcnt = 0;
629:                            strcpy (fncname, ftable[fncnt]);
630:                            fncout ();
631:                            prosflag = TRUE;
632:                            printf (" -Processing the %s function..¥n",fncname);
633:                            break;
634:                    }
635:
636:                    case EFUNC:
637:                    {
638:                            fncnt++;
639:                            fprintf (fpout, "%s$QED:¥n",fncname);
640:                            fprintf (fpout, "¥tDW¥t%d¥n",relcnt);
641:                            for (i = 0; i < relcnt; i++)
642:                            {
643:                                    fprintf (fpout, "¥tDW¥t%s$R%03d-%s$QSR-2¥n"
644:                                                    ,fncname,i,fncname);
645:                            }
646:                            fprintf (fpout,"¥n");
647:                            prosflag = FALSE;
648:                            break;
649:                    }
650:
651:                    case INCL:
652:                    {
653:                            incset ();
654:                            break;
655:                    }
656:
657:                    case EXTN:      break;
658:
```

```
659:                default:        outline ();
660:        }
661: }
662:
663:
664: /*      Function.Externals output       */
665:
666: fncout()
667: {
668:        char    fword[20];
669:        char    lword[20];
670:        int     i;
671:
672:        fprintf (fpout,"%s$QBG:¥n",fncname);
673:
674:        for (i = fnct[fncnt].extn; etable[i][0] != 0; i++)
675:        {
676:                first ( etable[i], fword);
677:                last ( etable[i], lword);
678:                fprintf (fpout,"¥tDB¥t'%s','%s'+80H¥n",fword,lword);
679:        }
680:        fprintf (fpout, "¥tDB¥t0¥n");
681:        fprintf (fpout, "¥tDW¥t%s$QED-$-2¥n",fncname);
682:        fprintf (fpout, "%s$QSR:¥n",fncname);
683:
684:        if ( etable[fnct[fncnt].extn][0] != 0 )
685:        {
686:                fprintf (fpout, "¥t%s¥t%s$QMN-%s$QSR¥n",JPMNE,fncname,fncname);
687:                fprintf (fpout, "%s$R%03d:¥n",fncname,relcnt++);
688:
689:                for ( i = fnct[fncnt].extn; etable[i][0] != 0; i++)
690:                        fprintf (fpout, "%s$%s¥tEQU¥t$-%s$QSR¥n¥t%s¥t0¥n"
691:                                        ,fncname, etable[i],fncname,JPMNE);
692:        }
693:        fprintf (fpout, "¥n%s$QMN:¥n",fncname);
694: }
695:
696:
697: /*-----------------------------
698:        Check label
699:        if exist: return TRUE
700:        else    : return FALSE
701: -------------------------------*/
702:
703: char    cklabel(word)
```

```
704: char	*word;
705: {
706:		int	i;
707:		for ( i = fnct[fncnt].label; ltable[i][0] != 0; i++)
708:		{
709:			if (! strcmp (word, ltable[i]))		return (TRUE);
710:		}
711:		return (FALSE);
712: }
713:
714:
715: /*---	Include set	---*/
716:
717: incset()
718: {
719: char	buf[TOKLEN];
720: char	*wordad;
721:
722:		if ( incflag )
723:		{
724:			printf (INCERR1);
725:			kexit();
726:		}
727:		gettoken (word);
728:		if (*word == '"')
729:		{
730:			wordad = word;
731:			wordad++;
732:			butlast (wordad,buf);
733:			strcpy (incname,buf);
734:		}
735:		else if (*word == '<')
736:		{
737:			gettoken (word);
738:			wordad = word;
739:			if ( *(++wordad) == ':')	++wordad;
740:			else				--wordad;
741:			strcpy (incname,wordad);
742:		}
743:
744:		if ( ERROR == fopen (incname, ifpin))
745:		{
746:			printf (INCERR2,incname);
747:			kexit();
748:		}
```

```
749:         incflag = TRUE;
750:         fpin = ifpin;
751: }
752:
753:
754: /*      Output file     */
755:
756: outline()
757: {
758:         rflag = FALSE;
759:         eflag = FALSE;
760:         jrflag = FALSE;
761:
762:         if ( cklabel (word))
763:         {                               /* when 1st token = label       */
764:
765:                 fprintf ( fpout,"%s$%s¥tEQU¥t$-%s$QSR¥n"
766:                                         ,fncname,word,fncname);
767:                 if (gettoken (word))
768:                 {
769:                         if (*ltokenpnt == ':' ) ltokenpnt++;
770:                         lineout();
771:                 }
772:                 else if (*ltokenpnt == ':' ) ltokenpnt++;
773:         }
774:         else    lineout();
775:
776:         fprintf ( fpout,"%s¥n",ltokenpnt );
777:         if (rflag && !eflag && !jrflag)
778:                 fprintf (fpout, "%s$R%03d:¥n",fncname,relcnt++);
779: }
780:
781:
782: lineout()
783: {
784:         do
785:         {
786:                 if ( cklabel (word))
787:                 {
788:                         outsp (ltokenpnt);
789:                         fprintf (fpout,"%s$%s",fncname,word);
790:                         rflag = TRUE;
791:                         if (jrflag)
792:                                 fprintf (fpout,"+%s$QSR",fncname);
793:                 }
```

```
794:
795:                    else
796:                    {
797:                            outsp (ltokenpnt);
798:                            fprintf (fpout,"%s",word);
799:                            if ( !strcmp (word,"EQU"))
800:                                    eflag = TRUE;
801: #if Z80
802:                            else if ( !strcmp (word,"JR") || !strcmp (word,"DJNZ"))
803:                                    jrflag = TRUE;
804: #endif
805:
806:                    }
807:            }
808:            while ( gettoken (word));
809: }
810:
811:
812:
813:
814: /*      Output space & comment  */
815:
816: outsp(pnt)
817: char    *pnt;
818: {
819:            while ( *pnt == ' ' || *pnt == '¥t' || *pnt == ':')
820:                    putc( *pnt++, fpout);
821: }
822:
823:
824: ismnem(word)
825: char    *word;
826: {
827:            int     i;
828:            switch ( strlen (word))
829:            {
830:                    case 2:
831:                    {
832:                            for ( i = 0; i < NEM2; i++)
833:                                    if (!strcmp (word, nem2[i]))
834:                                            return (TRUE);
835:                            break;
836:                    }
837:
838:                    case 3:
```

```
839:                    {
840:                            for ( i = 0; i < NEM3; i++)
841:                                    if (!strcmp (word, nem3[i]))
842:                                            return (TRUE);
843:                            break;
844:                    }
845:
846:                    case 4:
847:                    {
848:                            for ( i = 0; i < NEM4; i++)
849:                                    if (!strcmp (word, nem4[i]))
850:                                            return (TRUE);
851:                            break;
852:                    }
853:
854:                    default:;
855:            }
856:            for ( i = 0; i < SUBO; i++)
857:                    if (!strcmp (word, subo[i]))    return (TRUE);
858:
859:            if ( *word == '.' )     return (TRUE);
860:            return (FALSE);
861: }
862:
863:
864: /*---    initial routine ---
865:         nmemonic table set      */
866:
867: initdim()
868: {
869: #if Z80
870:         nem2[0] = "CP";
871:         nem2[1] = "DI";
872:         nem2[2] = "EI";
873:         nem2[3] = "IM";
874:         nem2[4] = "IN";
875:         nem2[5] = "JP";
876:         nem2[6] = "JR";
877:         nem2[7] = "LD";
878:         nem2[8] = "OR";
879:         nem2[9] = "RL";
880:         nem2[10] = "RR";
881:         nem2[11] = "EX";
882:         nem2[12] = "DB";
883:         nem2[13] = "DC";
```

```
884:        nem2[14] = "DS";
885:        nem2[15] = "DW";
886:
887:        nem3[0] = "ADC";
888:        nem3[1] = "ADD";
889:        nem3[2] = "AND";
890:        nem3[3] = "BIT";
891:        nem3[4] = "CCF";
892:        nem3[5] = "CPD";
893:        nem3[6] = "CPI";
894:        nem3[7] = "CPL";
895:        nem3[8] = "DAA";
896:        nem3[9] = "DEC";
897:        nem3[10] = "EXX";
898:        nem3[11] = "INC";
899:        nem3[12] = "IND";
900:        nem3[13] = "INI";
901:        nem3[14] = "LDD";
902:        nem3[15] = "LDI";
903:        nem3[16] = "NEG";
904:        nem3[17] = "NOP";
905:        nem3[18] = "OUT";
906:        nem3[19] = "POP";
907:        nem3[20] = "RES";
908:        nem3[21] = "RET";
909:        nem3[22] = "RLA";
910:        nem3[23] = "RRA";
911:        nem3[24] = "RLC";
912:        nem3[25] = "RRC";
913:        nem3[26] = "RLD";
914:        nem3[27] = "RRD";
915:        nem3[28] = "RST";
916:        nem3[29] = "SBC";
917:        nem3[30] = "SCF";
918:        nem3[31] = "SET";
919:        nem3[32] = "SLA";
920:        nem3[33] = "SRA";
921:        nem3[34] = "SRL";
922:        nem3[35] = "SUB";
923:        nem3[36] = "XOR";
924:
925:        nem3[37] = "EQU";
926:        nem3[38] = "ORG";
927:        nem3[39] = "END";
928:
```

```
929:
930:            nem4[0] = "CALL";
931:            nem4[1] = "CPDR";
932:            nem4[2] = "CPIR";
933:            nem4[3] = "DJNZ";
934:            nem4[4] = "HALT";
935:            nem4[5] = "INDR";
936:            nem4[6] = "INIR";
937:            nem4[7] = "LDDR";
938:            nem4[8] = "LDIR";
939:            nem4[9] = "OUTD";
940:            nem4[10] = "OTDR";
941:            nem4[11] = "OUTI";
942:            nem4[12] = "OTIR";
943:            nem4[13] = "PUSH";
944:            nem4[14] = "RETI";
945:            nem4[15] = "RETN";
946:            nem4[16] = "RLCA";
947:            nem4[17] = "RRCA";
948:
949:            nem4[18] = "DEFB";
950:            nem4[19] = "DEFS";
951:            nem4[20] = "DEFW";
952:            nem4[21] = "PAGE";
953:            nem4[22] = "ENDM";
954:
955: #else
956:            nem2[0] = "CC";
957:            nem2[1] = "CM";
958:            nem2[2] = "CP";
959:            nem2[3] = "CZ";
960:            nem2[4] = "DI";
961:            nem2[5] = "EI";
962:            nem2[6] = "IN";
963:            nem2[7] = "JC";
964:            nem2[8] = "JM";
965:            nem2[9] = "JP";
966:            nem2[10] = "JZ";
967:            nem2[11] = "RC";
968:            nem2[12] = "RM";
969:            nem2[13] = "RP";
970:            nem2[14] = "RZ";
971:
972:            nem2[15] = "DB";
973:            nem2[16] = "DS";
```

```
 974:          nem2[17] = "DW";
 975:
 976:          nem3[0] = "ACI";
 977:          nem3[1] = "ADC";
 978:          nem3[2] = "ADD";
 979:          nem3[3] = "ADI";
 980:          nem3[4] = "ANA";
 981:          nem3[5] = "ANI";
 982:          nem3[6] = "CMA";
 983:          nem3[7] = "CMC";
 984:          nem3[8] = "CMP";
 985:          nem3[9] = "CNC";
 986:          nem3[10] = "CNZ";
 987:          nem3[11] = "CPE";
 988:          nem3[12] = "CPI";
 989:          nem3[13] = "CPO";
 990:          nem3[14] = "DAA";
 991:          nem3[15] = "DAD";
 992:          nem3[16] = "DCR";
 993:          nem3[17] = "DCX";
 994:          nem3[18] = "HLT";
 995:          nem3[19] = "INR";
 996:          nem3[20] = "INX";
 997:          nem3[21] = "JMP";
 998:          nem3[22] = "JNC";
 999:          nem3[23] = "JNZ";
1000:          nem3[24] = "JPE";
1001:          nem3[25] = "JPO";
1002:          nem3[26] = "LDA";
1003:          nem3[27] = "LXI";
1004:          nem3[28] = "MOV";
1005:          nem3[29] = "MVI";
1006:          nem3[30] = "NOP";
1007:          nem3[31] = "ORA";
1008:          nem3[32] = "ORI";
1009:          nem3[33] = "OUT";
1010:          nem3[34] = "POP";
1011:          nem3[35] = "RAL";
1012:          nem3[36] = "RAR";
1013:          nem3[37] = "RET";
1014:          nem3[38] = "RLC";
1015:          nem3[39] = "RNC";
1016:          nem3[40] = "RNZ";
1017:          nem3[41] = "RPE";
1018:          nem3[42] = "RPO";
```

```
1019:         nem3[43] = "RRC";
1020:         nem3[44] = "RST";
1021:         nem3[45] = "SBB";
1022:         nem3[46] = "SBI";
1023:         nem3[47] = "STA";
1024:         nem3[48] = "STC";
1025:         nem3[49] = "SUB";
1026:         nem3[50] = "SUI";
1027:         nem3[51] = "XRA";
1028:         nem3[52] = "XRI";
1029:
1030:         nem3[53] = "EQU";
1031:         nem3[54] = "ORG";
1032:         nem3[55] = "END";
1033:         nem3[56] = "SET";
1034:
1035:
1036:         nem4[0] = "CALL";
1037:         nem4[1] = "LDAX";
1038:         nem4[2] = "LHLD";
1039:         nem4[3] = "PCHL";
1040:         nem4[4] = "PUSH";
1041:         nem4[5] = "SHLD";
1042:         nem4[6] = "SPHL";
1043:         nem4[7] = "STAX";
1044:         nem4[8] = "XCHG";
1045:         nem4[9] = "XTHL";
1046: #endif
1047:         subo[0] = "IF";
1048:         subo[1] = "ENDIF";
1049:         subo[2] = "ENTRY";
1050:         subo[3] = "GROBAL";
1051:         subo[4] = "PUBLIC";
1052:         subo[5] = "$EJECT";
1053:         subo[6] = "TITLE";
1054:         subo[7] = "MACRO";
1055:
1056: }
```

コラム2
〈CRLファイルの逆アセンブル〉

BDS C，α-Cはコンパイルとリンクという2つの処理だけで実行可能ファイルを生成するコンパイラですが，実際には多くのコンパイラはいったんアセンブラのソースプログラムを出力し，それをアセンブルするという手順を踏みます．

これは，得られたアセンブラプログラムを変更しやすい，アセンブラプログラムとの合成が容易であるなどの特徴がありますが，当然パスが増え，かつテンポラリ（一時）ファイルを読み書きするためコンパイル時間が長くかかるという欠点を持っています．

BDS Cは実用性を追及したコンパイラですから，このような方法はとっていませんが，時にはコンパイル結果のアセンブラリストが欲しい場合があります．これは主に，

1. プログラムオブジェクトを解析する場合
2. オブジェクトコードの内容を変更したい場合
3. 他のリンク形式のオブジェクトファイルを作りたい場合

などです．この目的のプログラムは特に用意されていませんが，BDS Cを使い込んでいくうちにどうしても必要になってきます．

そこで，CRLファイルを逆アセンブルし，アセンブラのソースファイルに変換するプログラムを作成しました．しかし，残念ながら本書の執筆に間に合わず，本文中で紹介する事ができませんでした．そこで本書のディスクサービス（巻末参照）を利用し，本書の読者にお分けすることにしました．

DACRLと名付けたこのプログラムはZ80ニーモニックの逆アセンブラで，CRLファイルをZCASMのソースプログラムに変換します．従って，細部を変更後，再びZCASMを使って，CRLファイルに戻す事ができます．外部関数の呼出し，文字列などもラベルを付加し正しく変換されます．ZCASMとともにBDS Cを本格的に使う際にはどうしても必要なソフトウェアでしょう．

第 6 章

BDS C技術情報

BDS　Cは8ビット用のC言語としては標準的なものといえますが，専用のリンク形式(CRLフォーマット）を持った，かなり独立性の高いソフトウェアです．他のC言語と同じく，本格的に大きなプログラムを作成する場合には知っておいた方が良い事，知っていなければならない事が数多くあります．

本章では，プログラムを作成する場合に役立つ，比較的高度な内容について述べます．

6-1　CRLフォーマット

CP／Mではリロケータブルなオブジェクトファイルを作成する場合，マイクロソフト社のMACRO-80などが出力するファイルフォーマット(RELファイル）を使うのが一般的ですが，BDS CのCRLフォーマットはこれとは全く異なっています．

CRLファイルの構造はマニュアルにも詳しく述べられていますので，ここでは特に重要な点について説明します．

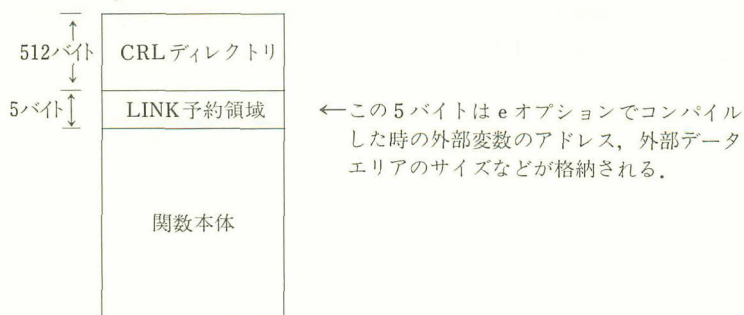



図6-1　CRLファイル構造

図6-1は全体的なファイルの構造を示しています．この中で，CRLディレクトリはどんなCRLファイルであっても512バイトに固定されており，これが一つのCRLファイル内に格納可能な関数の数の制限の一つになっています．ディレクトリは図6-2のように構成されています．

図6-2　CRLディレクトリの構造

関数名は8文字までとなっていますので，ポインタ2バイトとで1関数当り最大10バイト必要としますが，1つのCRLファイルは最大63関数までに制限されています．なお，8文字以上の名前を持つ関数を作ると頭の8文字だけが有効となり，他のCRLファイルとリンクする時に困る事がありますので注意してください．

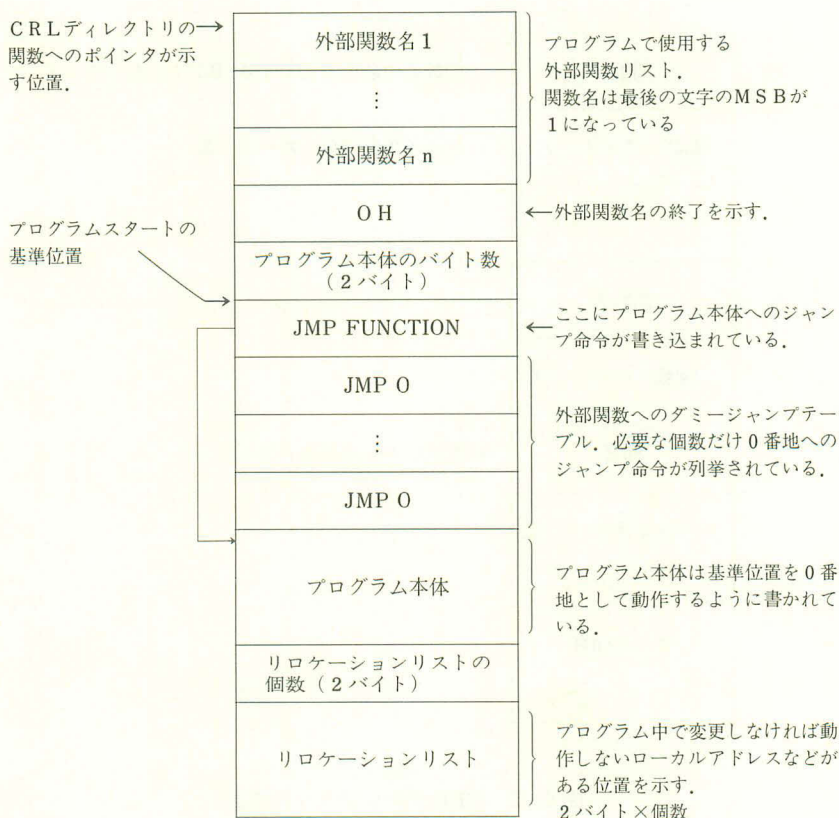



図６－３　関数本体の構造

図６－３は関数の本体で，かなり複雑です．プログラムは基準アドレスに対してすべてリロケートされて作成されますが，外部関数名，プログラムサイズに続いて，プログラム本体へのジャンプ命令があり，次いで外部関数へのダミージャンプテーブルが続きます．このダミージャンプテーブルは０番地へのジャンプ命令(JMP　0)の羅列ですが，リンク時に関数名に従ってジャンプアドレスが書き換えられます．一般に外部の関数を呼び出す時は，すべてこのテーブルを介してジャンプします．

ただ，このジャンプテーブルは，プログラムの実行そのものには必要なく，

単にリンクが簡単になるように加えられているものですから，無駄な部分でもあります．ここがL2では改善されており，L2でリンクした場合には，ジャンプテーブルは取り除かれて外部関数をプログラム中からじかに呼び出すように変更されます．従ってオブジェクトは小さくなり，実行もやや早くなります．なるべくL2を利用する方が有利です（リンク時間はCLINKより長くかかります）．

最後にあるのは，リロケーションリストです．これは，プログラムが正しく動作するようにリンク時に書き換えなければならない位置を示しています．（主にJMP，CALL命令などのアドレスです）．

CRLファイルの構造はDDTなどによって内部をチェックしてみると比較的わかりやすいと思いますが，CASM，ZCASMによって作成されたアセンブラソースリストを見るのも参考になるでしょう．

6-2　発生したオブジェクトの内容

コンパイラによって発生する機械語オブジェクトは，Cソースプログラムの書き方によってほぼ定まってきます．ここで紹介するものはBDS Cの持つ機能の一部でしかありませんが，これらを知識として知っておくだけでも小さく，早い実行プログラムを作る上でかなり役に立つでしょう．

なお，テストは小さなサンプルプログラムを作成してコンパイルし，L2でリンクした実行ファイルを逆アセンブルしておこないました．従って実行アドレスが記述されていますが，これらは無視して考えてください．

また本項では逆アセンブルリストをザイログZ80ニーモニックとインテル8080ニーモニックの2つで行なっています．左側がザイログ，右側がインテルニーモニックです．

本論にはいる前に基本的なレジスタ使用法について述べておきます．

まず，BDS Cでは演算やアドレスなどのアキュームレータとしてHLレジスタを用いています．演算に必要なディスティネーションレジスタにはDEレジスタを使用しますので，加算などでは

ADD　HL, DE　|　DAD　D

という命令が使われます.

また，BCレジスタは，CASMの項でも述べたようにローカル変数の先頭を示すフレームポインタとして使われますので，アセンブラ定義関数以外では一般の処理に用いられる事はありません．BDS Cは16ビット処理が基本ですから，Aレジスタは補助的に使われるに過ぎませんがchar型変数の処理においては積極的に用いられます.

退避などにはPUSH, POP命令でスタックを利用しています．これらは一般にアセンブラでプログラムを作成する場合と同じです.

なお，BDS Cで大きな特徴と思われるのはソースプログラムを詳細にチェックし最適化(オプティマイズ)がかなりおこなわれている事です．このためオブジェクトは非常に洗練されています.

(1) ローカル変数と外部変数

ローカル変数は，C言語の特張でもあり欠かす事のできない機能ですが，残念ながら8080(Z80)CPUでは簡単に実現できません.

そこでBDS Cはスタック上に領域をとり，ここをローカル変数として利用しています．しかし，参照に要するオーバーヘッドはかなり大きなものとなっています.

リスト6-4　ローカル変数の参照

```
main()
{
        int     a,b;   ←―――― ローカル変数の宣言.
        a = a+b;          /* add  */
}
                  〈ザイログニー          〈インテルニー
                      モニック〉              モニック〉

08B8   C5            PUSH    BC            PUSH B
08B9   21 FC FF      LD      HL,-4         LXI  H,-4
```

```
08BC   39         ADD    HL,SP          DAD  SP       上位関数のフレームポインタ
08BD   F9         LD     SP,HL          SPHL          の退避とこの関数で用いる
08BE   44         LD     B,H            MOV  B,H      フレームポインタの設定.
08BF   4D         LD     C,L            MOV  C,L

08C0   CD 90 01   CALL   0190 [SDLI]    CALL 0190     a
08C3   00         DB     0              DB   0
08C4   E5         PUSH   HL             PUSH H
08C5   CD 90 01   CALL   0190 [SDLI]    CALL 0190     b
08C8   02         DB     2              DB   2
08C9   D1         POP    DE             POP  D
08CA   19         ADD    HL,DE          DAD  D        a+b→HL

08CB   EB         EX     DE,HL          XCHG
08CC   60         LD     H,B            MOV  H,B
08CD   69         LD     L,C            MOV  L,C      加算結果を
08CE   73         LD     (HL),E         MOV  M,E      変数aに代入.
08CF   23         INC    HL             INX  H
08D0   72         LD     (HL),D         MOV  M,D

08D1   EB         EX     DE,HL          XCHG
08D2   21 04 00   LD     HL,0004        LXI  H,0004
08D5   39         ADD    HL,SP          DAD  SP
08D6   F9         LD     SP,HL          SPHL          フレームポインタの復帰と
08D7   EB         EX     DE,HL          XCHG          リターン.
08D8   C1         POP    BC             POP  B
08D9   C9         RET                   RET
```

リスト6-4で明らかなように,関数中でローカル変数を参照する場合にはまずBCレジスタにフレームポインタのセットを行ないます. このフレームポインタのセットは関数の呼出し毎に行なわれるのでかなり大きなオーバーヘッドとなりますが, ローカル変数を使わず, 引数も無い場合には最適化により省略されます(リスト6-5, 6-6参照).

変数の値の参照にはランタイムパッケージ(C.CCC)の中にあるSDLI(short-displacement, double-byte local indirection)というサブルーチンをCALLして行ないます(コンパイルオプション“-o”のない時).

このサブルーチンは,

```
CALL    SDLI
DB      n        (offset)
```

という形で呼び出され，nの値によってフレームポインタ（BC)からnバイトの位置にあるメモリの内容(２バイト)をHLレジスタに入れ，DBの次の番地に返ってきます．ローカル変数だけでなく引数の値を参照する場合にも用いられます．

なお,このサブーチンではフレームポインタから255バイトめの変数までしか参照できませんので，それ以上距離のある変数を見たい場合には，LDLIを用い，

```
CALL    LDLI
DW      nn       (offset)
```

という形で参照します．

変数への書込みはサブルーチンではなく，オブジェクト上でアドレスを計算して行なわれます．

リスト6-5　外部変数の参照1

```
int     a,b;  ←―――――――――――――――― 外部変数の宣言.
main()
{
        a = a+b;        /* add  */
}
                    〈ザイログニー              〈インテルニー
                      モニック〉                 モニック〉

08B8   C5         PUSH   BC               PUSH B

08B9   CD 5C 01   CALL   015C [SDEI]      CALL 015C  } a
08BC   00         DB     0                DB   0     }
08BD   E5         PUSH   HL               PUSH H
08BE   CD 5C 01   CALL   015C [SDEI]      CALL 015C  } b
08C1   02         DB     2                DB   2     }
08C2   D1         POP    DE               POP  D
08C3   19         ADD    HL,DE            DAD  D     } a+b→HL
```

```
08C4   E5          PUSH   HL                   PUSH H
08C5   2A 15 01    LD     HL,(0115 [EXTRNS])   LHLD 0115
08C8   D1          POP    DE                   POP  D
08C9   73          LD     (HL),E               MOV  M,E
08CA   23          INC    HL                   INX  H
08CB   72          LD     (HL),D               MOV  M,D

08CC   C1          POP    BC                   POP  B
08CD   C9          RET                         RET
```

（08C4〜08CB：加算結果をaに代入.）

外部変数の場合も参照の様子はほぼ同じです．リスト6-5ではローカル変数も引数もありませんので，フレームポインタのセットをしていませんが，変数のアクセス法は基本的に同じです．

外部変数エリアの先頭アドレスはランタイムパッケージ中の変数 EXTRNS（115番地）に格納されていますので，書込みの場合にはサブルーチンSDEI（short-displacement double byte external indirection），あるいはLDEIを用います．詳細については次のランタイムパッケージの項を参照してください．

リスト6-6　外部変数の参照2

```
int     a,b;  ←――――――――――――――――― 外部変数宣言.
main()
{
        a = a+b;        /* add  */
}
                    〈ザイログニー                〈インテルニー
                       モニック〉                    モニック〉

08B8   C5          PUSH   BC           PUSH B

08B9   2A 00 10    LD     HL,(1000)    LHLD 1000   ←a
08BC   E5          PUSH   HL           PUSH H      ←b
08BD   2A 02 10    LD     HL,(1002)    LHLD 1002
08C0   D1          POP    DE           POP  D
08C1   19          ADD    HL,DE        DAD  D      ←HL=a+b
08C2   22 00 10    LD     (1000),HL    SHLD 1000   ←a=HL

08C5   C1          POP    BC           POP  B
08C6   C9          RET                 RET
```

※オプション－e1000でコンパイル.

さて，リスト6-6はコンパイル時にオプションeを指定してコンパイルした場合です．この時は外部変数のアドレスがわかり，

LD HL, (nn)	LHLD nn
LD (nn), HL	SHLD nn

が使えるので，オブジェクトが小さくなります．

この効果はかなり顕著ですから，少なくとも完成したプログラムは－eオプションをつけてコンパイルした方が有利です．

(2) インクリメント，デクリメントを用いた最適化

変数に1や2を加えたり減じたりする処理は非常に頻繁に行なわれるものですが，BDS Cではこの時かなり思い切った最適化が行なわれます．

リストをまずご覧ください．

リスト6-7 加算のテスト1

```
int     a;  ←── 外部変数.
main()
{
        a = a+5;
        a = a+4;
        a = a+1;
        a += 1;
        a++;
        ++a;
}
```

〈ザイログニーモニック〉 〈インテルニーモニック〉

```
08B8  C5           PUSH  BC            PUSH B

08B9  2A 00 10     LD    HL,(1000)     LHLD 1000   ┐
08BC  11 05 00     LD    DE,0005       LXI  D,0005 │ a=a+5;
08BF  19           ADD   HL,DE         DAD  D      │
08C0  22 00 10     LD    (1000),HL     SHLD 1000   ┘

08C3  2A 00 10     LD    HL,(1000)     LHLD 1000
08C6  23           INC   HL            INX  H
```

```
08C7  23        INC  HL          INX  H     } a=a+4;
08C8  23        INC  HL          INX  H
08C9  23        INC  HL          INX  H
08CA  22 00 10  LD   (1000),HL   SHLD 1000

08CD  2A 00 10  LD   HL,(1000)   LHLD 1000  } a=a+1;
08D0  23        INC  HL          INX  H
08D1  22 00 10  LD   (1000),HL   SHLD 1000

08D4  2A 00 10  LD   HL,(1000)   LHLD 1000  } a+=1;
08D7  23        INC  HL          INX  H
08D8  22 00 10  LD   (1000),HL   SHLD 1000

08DB  2A 00 10  LD   HL,(1000)   LHLD 1000  } a++;
08DE  23        INC  HL          INX  H
08DF  22 00 10  LD   (1000),HL   SHLD 1000

08E2  2A 00 10  LD   HL,(1000)   LHLD 1000  } ++a;
08E5  23        INC  HL          INX  H
08E6  22 00 10  LD   (1000),HL   SHLD 1000

08E9  C1        POP  BC          POP  B
08EA  C9        RET              RET
```

※オプション－e1000でコンパイル．

リスト6-7は－eオプションをつけてコンパイルし，外部変数を用いた場合ですが，歴然としているように，a＝a＋nなどの自分自身に定数を加算する時には4までの値であればインクリメントでおこなわれます．3まではオブジェクトサイズを小さくする効果がありますが，4では加算を用いてもバイト数は同じでどちらでも良いようなものですが，インクリメント命令は高速なのでこちらの方がスピード的に有利なのです．これは減算の際にも，ローカル変数を用いた場合も同じです．

特に注目しなければならないのはa＝a＋1，a＋＝1，a＋＋，＋＋aの4つの場合です．リストから明らかなようにこれらのオブジェクトは完全に同じです．一般にC言語では，＋＝，＋＋という演算子は高速なオブジェクトを発生する目的でわざわざ用意されているものですから，a＝a＋1と書くよりもa＋＝1，a＋＝1と書くよりもa＋＋と書く方が高速で小さなオブジェクトが発生するというのがC言語の常識なので，意外な気がします．

しかし，これはオプションeコンパイルの時の外部変数にのみ言える事で，ローカル変数の場合には，リスト6-8のようなオブジェクトが生成されます．

リスト6-8　加算のテスト2

```
main()
{
        int     a;   ←── ローカル変数.
        a = a+1;
        a += 1;
        a++;
}
                        <ザイログニー              <インテルニー
                           モニック>                  モニック>

08B8   C5          PUSH   BC              PUSH B       ┐
08B9   21 FE FF    LD     HL,-2           LXI  H,-2    │
08BC   39          ADD    HL,SP           DAD  SP      │ フレームポインタの設定,
08BD   F9          LD     SP,HL           SPHL         │
08BE   44          LD     B,H             MOV  B,H     │
08BF   4D          LD     C,L             MOV  C,L     ┘

08C0   CD 90 01    CALL   0190 [SDLI]     CALL 0190    ┐
08C3   00          DB     0               DB   0       │
08C4   23          INC    HL              INX  H       │
08C5   EB          EX     DE,HL           XCHG         │
08C6   60          LD     H,B             MOV  H,B     │ a=a+1
08C7   69          LD     L,C             MOV  L,C     │
08C8   73          LD     (HL),E          MOV  M,E     │
08C9   23          INC    HL              INX  H       │
08CA   72          LD     (HL),D          MOV  M,D     ┘

08CB   60          LD     H,B             MOV  H,B     ┐
08CC   69          LD     L,C             MOV  L,C     │
08CD   E5          PUSH   HL              PUSH H       │
08CE   7E          LD     A,(HL)          MOV  A,M     │
08CF   23          INC    HL              INX  H       │
08D0   66          LD     H,(HL)          MOV  H,M     │
08D1   6F          LD     L,A             MOV  L,A     │ a+=1
08D2   23          INC    HL              INX  H       │
08D3   EB          EX     DE,HL           XCHG         │
08D4   E1          POP    HL              POP  H       │
08D5   73          LD     (HL),E          MOV  M,E     │
08D6   23          INC    HL              INX  H       │
08D7   72          LD     (HL),D          MOV  M,D     ┘
```

```
08D8   60          LD    H,B             MOV  H,B     ┐
08D9   69          LD    L,C             MOV  L,C     │
08DA   5E          LD    E,(HL)          MOV  E,M     │
08DB   23          INC   HL              INX  H       │
08DC   56          LD    D,(HL)          MOV  D,M     ├ a++
08DD   13          INC   DE              INX  D       │
08DE   72          LD    (HL),D          MOV  M,D     │
08DF   2B          DEC   HL              DCX  H       │
08E0   73          LD    (HL),E          MOV  M,E     ┘

08E1   EB          EX    DE,HL           XCHG         ┐
08E2   21 02 00    LD    HL,0002         LXI  H,0002  │
08E5   39          ADD   HL,SP           DAD  SP      │
08E6   F9          LD    SP,HL           SPHL         ├ フレームポインタの復活.
08E7   EB          EX    DE,HL           XCHG         │
08E8   C1          POP   BC              POP  B       │
08E9   C9          RET                   RET          ┘
```

リスト6-8では，C言語の常識通りにオブジェクトが発生しています．a++が一番バイト数が少なくなっており，処理も高速です．しかしa＝a＋1はバイト数は少ないもののSDLIをCALLするため処理が遅くなっています．そこで，コンパイル時に－oオプションをつけてコンパイルすると，次のように展開されます．

リスト6-9　加算のテスト3

```
main()
{
        int     a;◄──────────────────────────── ローカル変数.
        a = a+1;
}
                    〈ザイログニー              〈インテルニー
                        モニック〉                  モニック〉

08B8   C5          PUSH  BC              PUSH B       ┐
08B9   21 FE FF    LD    HL,-2           LXI  H,-2    │
08BC   39          ADD   HL,SP           DAD  SP      ├ フレームポインタの設定.
08BD   F9          LD    SP,HL           SPHL         │
08BE   44          LD    B,H             MOV  B,H     │
08BF   4D          LD    C,L             MOV  C,L     ┘

08C0   60          LD    H,B             MOV  H,B     ┐
08C1   69          LD    L,C             MOV  L,C     │
08C2   7E          LD    A,(HL)          MOV  A,M     ├ SDLIの展開.
```

```
08C3  23        INC  HL          INX  H       ┐                    ┐
08C4  66        LD   H,(HL)      MOV  H,M     │                    │
08C5  6F        LD   L,A         MOV  L,A     ┘                    │
08C6  23        INC  HL          INX  H       +1                   │ a=a+1;
08C7  EB        EX   DE,HL       XCHG         ┐                    │
08C8  60        LD   H,B         MOV  H,B     │                    │
08C9  69        LD   L,C         MOV  L,C     │ 後半は全く同じ.     │
08CA  73        LD   (HL),E      MOV  M,E     │                    │
08CB  23        INC  HL          INX  H       │                    │
08CC  72        LD   (HL),D      MOV  M,D     ┘                    ┘

08CD  EB        EX   DE,HL       XCHG         ┐
08CE  21 02 00  LD   HL,0002     LXI  H,0002  │
08D1  39        ADD  HL,SP       DAD  SP      │
08D2  F9        LD   SP,HL       SPHL         │ フレームポインタの復活.
08D3  EB        EX   DE,HL       XCHG         │
08D4  C1        POP  BC          POP  B       │
08D5  C9        RET              RET          ┘
```

※オプション－oでコンパイル

SDLIが直接コードに置き換えられているのがわかります．オプション－oは処理を高速化する最適化を指定するものですが，具体的な内容はこのようにローカル変数のアクセス法を変更しているのです．

なお，この時，SDLIが

```
LD    H, B        |  MOV   H, B
LD    L, C        |  MOV   L, C
LD    A, (HL)     |  MOV   A, M
INC   HL          |  INX   H
LD    H, (HL)     |  MOV   H, M
LD    L, A        |  MOV   L, A
```

と展開されていますが，これは，たまたま先頭のローカル変数のアクセスだからで，一般には，最初の2命令は，

```
LD   HL, nn    |  LXI  H, nn
ADD  HL, BC    |  DAD  B
```

となります．最初に宣言したローカル変数のほうが処理が簡単ですむという事

にも注目すべきでしょう.

次に一致などの場合を見てみましょう.

リスト 6-10 if文のテスト

```
int     a,b;  ←――――――――――――――――――――――――― 外部変数.
main()
{
        if (a == 5)     b++;
        if (a == 4)     b++;
        if (a == 0)     b++;
        if (!a)         b++;
}
                   〈ザイログニー            〈インテルニー
                      モニック〉                モニック〉

08B8  C5          PUSH  BC            PUSH B
08B9  2A 00 10    LD    HL,(1000)     LHLD 1000   ┐
08BC  11 FB FF    LD    DE,FFFB       LXI  D,FFFB │
08BF  19          ADD   HL,DE         DAD  D      │
08C0  7C          LD    A,H           MOV  A,H    ├ if(a==5)
08C1  B5          OR    A,L           ORA  L      │
08C2  C2 CC 08    JP    NZ,08CC       JNZ  08CC   ┘

08C5  2A 02 10    LD    HL,(1002)     LHLD 1002   ┐
08C8  23          INC   HL            INX  H      ├ b++
08C9  22 02 10    LD    (1002),HL     SHLD 1002   ┘

08CC  2A 00 10    LD    HL,(1000)     LHLD 1000   ┐
08CF  2B          DEC   HL            DCX  H      │
08D0  2B          DEC   HL            DCX  H      │
08D1  2B          DEC   HL            DCX  H      │
08D2  2B          DEC   HL            DCX  H      ├ if(a==4)
08D3  7C          LD    A,H           MOV  A,H    │
08D4  B5          OR    A,L           ORA  L      │
08D5  C2 DF 08    JP    NZ,08DF       JNZ  08DF   ┘

08D8  2A 02 10    LD    HL,(1002)     LHLD 1002   ┐
08DB  23          INC   HL            INX  H      ├ b++
08DC  22 02 10    LD    (1002),HL     SHLD 1002   ┘

08DF  2A 00 10    LD    HL,(1000)     LHLD 1000   ┐
08E2  7C          LD    A,H           MOV  A,H    │
08E3  B5          OR    A,L           ORA  L      ├ if(a==0)
08E4  C2 EE 08    JP    NZ,08EE       JNZ  08EE   ┘
```

```
08E7   2A 02 10    LD     HL,(1002)        LHLD 1002   ┐
08EA   23          INC    HL               INX  H      ├ b++
08EB   22 02 10    LD     (1002),HL        SHLD 1002   ┘

08EE   2A 00 10    LD     HL,(1000)        LHLD 1000   ┐
08F1   7C          LD     A,H              MOV  A,H    │
08F2   B5          OR     A,L              ORA  L      ├ if ( ! a)
08F3   C2 FD 08    JP     NZ,08FD          JNZ  08FD   ┘

08F6   2A 02 10    LD     HL,(1002)        LHLD 1002   ┐
08F9   23          INC    HL               INX  H      ├ b++
08FA   22 02 10    LD     (1002),HL        SHLD 1002   ┘

08FD   C1          POP    BC               POP  B
08FE   C9          RET                     RET
```

※オプションーe1000でコンパイル.

リスト6-10をみても明らかなように, a == nなどの比較は最適化されてており, nが4まではDEC命令を用いて処理が高速化されています. これはa = a + nの場合とほぼ同様ですが注目しなければならないのは if (a == 0) とif(! a) です.

( )の中を真か偽かテストする時は, if (a != 0) より, if (a)の方が小さくなる, というのがC言語の常識ですから当然if (a == 0) の方がif(! a) より大きいと思うのが人情というものです. ところがBDS Cでは完全にこの書式が最適化されており, どちらの書式を用いても全く同じコードが作成されます. これはif (a ! = 0) とif (a) の場合にも同じです. 変数の種類にも関係しません.

従って, ifなどの条件式においては最適化によって, 常に最小のコードが発生しますのでオブジェクトの事など考えず, 最も分りやすい書式を選ぶのがBDS流と言えるでしょう

(3) 関数の呼出し

BDS Cでは, 関数を呼び出す前に引数をスタックにPUSHします. 呼出された関数ではローカル変数の場合と全く同じ方法でその内容を参照します.

この関数呼出しはオーバーヘッドが大きい部分ですが，書式によりそのオブジェクトの生成を制御する事はできないので，BDS Cの基本的な仕様の部分と言えるでしょう．

リスト6-11 引数の参照

```
main(a,b)
{
        return (a+b);
}
                      <ザイログニー              <インテルニー
                          モニック>                  モニック>

08B8  C5         PUSH  BC              PUSH B
08B9  21 00 00   LD    HL,0000         LXI  H,0000
08BC  39         ADD   HL,SP           DAD  SP
08BD  F9         LD    SP,HL           SPHL
08BE  44         LD    B,H             MOV  B,H
08BF  4D         LD    C,L             MOV  C,L

08C0  CD 90 01   CALL  0190 [SDLI]     CALL 0190  } 引数aの参照.
08C3  04         DB    4               DB   4     }
08C4  E5         PUSH  HL              PUSH H
08C5  CD 90 01   CALL  0190 [SDLI]     CALL 0190  } 引数bの参照.
08C8  06         DB    6               DB   6     }
08C9  D1         POP   DE              POP  D
08CA  19         ADD   HL,DE           DAD  D     ←(a+b→HL)
08CB  C1         POP   BC              POP  B     } return
08CC  C9         RET                   RET        }

08CD  C1         POP   BC              POP  B     } 関数の最後に必らず
08CE  C9         RET                   RET        } 発生するオブジェクト.
                                                    この関数では全く無駄な
                                                    部分になる.
```

リスト6-11を見ると，このように短い関数でもこれだけのオブジェクトが発生してしまいます．特に引数の参照は必ずSDLIへのCALLが行なわれます．SDLIを用いていますから，－oオプションをつけてコンパイルすればじかにオブジェクトが展開されますが，遅い，大きいという欠点には変りありません．

根本的な対策はありませんが，小さな関数はマクロを使う事で高速化を図る事ができる場合もあります．関数のマクロへの置き換えとしては，第4章で紹介した浮動小数点，倍精度演算用マクロが良い例になるでしょう。

(4) 条件式

if文などで条件式を記述する際，2つの条件の論理積(AND)をとる場合も多いものです．例えば，

```
if (a==0 && b>0)
```

という条件があった場合，BDS Cではもし第1条件が成立しなければ第2条件のチェックは行ないません．従って，a＝＝0とb＞0という条件のうちa＝＝0のほうが頻繁に成立するのであれば，

```
if (b>0 && a==0)
```

と記述した方が処理は速くなります．条件の数が3つ以上でも左から順にチェックしますから，早く除外できる条件程先に書くべきです．

なお，論理和(OR)の場合でも考え方は同じです．ORの時は成立しやすい条件を先に書きます．

(5) 定数の記述

```
a=1+2+3+4+b;
```

と記述するとコンパイラはコンパイル時に定数部分を計算し，

```
a=10+b;
```

と置き直してオブジェクトを生成します．ところが，

```
a=b+1+2+3+4;
```

とするとこのままのオブジェクトになってしまいます．定数式の場合，BDS Cはある条件のもとだけ最適化を行なっているからで，

```
[=(,case return
```

の記号などに続く定数式はあらかじめ計算します．しかし，この規則を覚え

るのは面倒な事ですから，不安な場合は定数式を(　)で囲ってしまうのが簡単です．上記の例では

a＝b＋(１＋2＋3＋4)；

とすれば最適化されます．

(6) switch文

switch文はC言語において高速で判りやすい記述を狙った構文ですが，どのようにオブジェクトが展開されているのか見てみましょう，

リスト6-12　switch文のテスト

```
char    a;   ←── 外部変数.
main()
{
        switch (a)
        {
                case 9:   a=0;
                          break;
                case 100: a=1;
                          break;
                default:;
        }
}
```

		〈ザイログニーモニック〉		〈インテルニーモニック〉	
08B8	C5	PUSH	BC	PUSH B	
08B9	2A 00 10	LD	HL,(1000)	LHLD 1000	
08BC	7D	LD	A,L	MOV A,L	case 9
08BD	FE 09	CP	A,09	CPI 09	
08BF	CA CA 08	JP	Z,08CA	JZ 08CA	
08C2	FE 64	CP	A,64	CPI 64	case 100
08C4	CA D2 08	JP	Z,08D2	JZ 08D2	
08C7	C3 DA 08	JP	08DA	JMP 08DA	
08CA	3E 00	LD	A,00	MVI A,00	
08CC	32 00 10	LD	(1000),A	STA 1000	
08CF	C3 DA 08	JP	08DA	JMP 08DA	
08D2	3E 01	LD	A,01	MVI A,01	
08D4	32 00 10	LD	(1000),A	STA 1000	
08D7	C3 DA 08	JP	08DA	JMP 08DA	

```
08DA    C1          POP    BC              POP  B    ←
08DB    C9          RET                    RET
```

※オプション－e1000でコンパイル.

リスト6-12はchar型の変数で2つのcaseを持つ例です.変数aの内容をAレジスタにセットし，case文の数だけ最初にジャッジしています.

リスト6-13　if文のテスト

```
char    a;←――――――――――――――――――――――――――――――――――――― 外部変数.
main()
{
        if (a==9)        a=0;
        else if (a==100) a=1;
}
                       <ザイログニー            <インテルニー
                          モニック>               モニック>

08B8    C5          PUSH   BC              PUSH B

08B9    2A 00 10    LD     HL,(1000)       LHLD 1000  ┐
08BC    7D          LD     A,L             MOV  A,L   │
08BD    FE 09       CP     A,09            CPI  09    │
08BF    C2 CA 08    JP     NZ,08CA         JNZ  08CA  ├ if(a==9)  a=0;
08C2    3E 00       LD     A,00            MVI  A,00  │
08C4    32 00 10    LD     (1000),A        STA  1000  │
08C7    C3 D8 08    JP     08D8            JMP  08D8  ┘

08CA    2A 00 10    LD     HL,(1000)       LHLD 1000  ┐
08CD    7D          LD     A,L             MOV  A,L   │
08CE    FE 64       CP     A,64            CPI  64    │
08D0    C2 D8 08    JP     NZ,08D8         JNZ  08D8  ├ if(a==100)  a=1;
08D3    3E 01       LD     A,01            MVI  A,01  │
08D5    32 00 10    LD     (1000),A        STA  1000  ┘

08D8    C1          POP    BC              POP  B
08D9    C9          RET                    RET
```

※オプション－e1000でコンパイル.

リスト6-13は全く同一のプログラムをif文で書いた例です. case文が2つしかありませんので，むしろプログラムのサイズはif文によるものの方が短くなっていますが，重要な事は，メモリアクセスの回数です. if文では，それぞれ条件式がありますから，その判定ごとに変数aを読み出しますがswitch

では条件はただ1つですから1回だけになっています．この例では変数aは－eオプションをつけた外部変数ですからたまたま簡単なアクセスで済んでいますが，ローカル変数や関数の引数であればもっと時間とバイト数がかかり，caseが多くなるごとにその差は広がっていきます．switch文が使える場合はなるべく利用する方が有利です（リスト6-12，6-13はかなり例外的な場合です）．

なお，switch文はその対象になる変数がint型になると2バイト一致の検出をしなければなりませんから，なるべくchar型で行なった方が良いでしょう．また，一致しやすい条件ほど最初の方に記述するべきです．

(7) 配例とポインタ

配例を読み書きする際にはなるべくポインタを使うようにしたほうがプログラムが小さく速くなります．

文字配列を読み書きする際に配列を使った例をリスト6-14に示します．

リスト6-14 配列のテスト

```
int     arycnt;

arprint(ary)
char    ary[];
{
        arycnt = 0;
        while (ary[arycnt])
                putchar (ary[arycnt++]);
}

main()
{
        arprint ("mitarai");
}
```

〈ザイログニーモニック〉　〈インテルニーモニック〉

```
ARPRINT:
08B8  C5          PUSH  BC        PUSH B
08B9  21 00 00    LD    HL,0000   LXI  H,0000
08BC  39          ADD   HL,SP     DAD  SP
08BD  F9          LD    SP,HL     SPHL
08BE  44          LD    B,H       MOV  B,H
08BF  4D          LD    C,L       MOV  C,L
```

フレームポインタの設定．

```
08C0   21 00 00    LD     HL,0000            LXI  H,0000  } arycnt = 0;
08C3   22 00 10    LD     (1000),HL          SHLD 1000    }

08C6   21 04 00    LD     HL,0004            LXI  H,0004  }
08C9   09          ADD    HL,BC              DAD  B       }
08CA   7E          LD     A,(HL)             MOV  A,M     }
08CB   23          INC    HL                 INX  H       } ary[ ]の先頭アドレス.
08CC   66          LD     H,(HL)             MOV  H,M     }
08CD   6F          LD     L,A                MOV  L,A     }
08CE   E5          PUSH   HL                 PUSH H       }

08CF   2A 00 10    LD     HL,(1000)          LHLD 1000    }
08D2   D1          POP    DE                 POP  D       } ary[arycnt]
08D3   19          ADD    HL,DE              DAD  D       }
08D4   6E          LD     L,(HL)             MOV  L,M     }
08D5   7D          LD     A,L                MOV  A,L     }
08D6   B7          OR     A,A                ORA  A       } while
08D7   CA F8 08    JP     Z,08F8             JZ   08F8    }

08DA   21 04 00    LD     HL,0004            LXI  H,0004  }
08DD   09          ADD    HL,BC              DAD  B       }
08DE   7E          LD     A,(HL)             MOV  A,M     } ary[ ]の
08DF   23          INC    HL                 INX  H       } 先頭アドレス.
08E0   66          LD     H,(HL)             MOV  H,M     }
08E1   6F          LD     L,A                MOV  L,A     }
08E2   E5          PUSH   HL                 PUSH H

08E3   2A 00 10    LD     HL,(1000)          LHLD 1000    }              }
08E6   23          INC    HL                 INX  H       } arycnt++     }
08E7   22 00 10    LD     (1000),HL          SHLD 1000    }              }
08EA   2B          DEC    HL                 DCX  H       }              }
08EB   D1          POP    DE                 POP  D                      } ary[arycnt++]
08EC   19          ADD    HL,DE              DAD  D                      }
08ED   6E          LD     L,(HL)             MOV  L,M                    }
08EE   26 00       LD     H,00               MVI  H,00                   }
08F0   E5          PUSH   HL                 PUSH H                      }

08F1   CD 0D 09    CALL   090D [PUTCHAR]     CALL 090D    } putchar と
08F4   D1          POP    DE                 POP  D       } while へのジャンプ
08F5   C3 C6 08    JP     08C6               JMP  08C6    }

08F8   C1          POP    BC                 POP  B
08F9   C9          RET                       RET
```

```
MAIN:
08FA   C5              PUSH   BC                PUSH B
08FB   21 05 09        LD     HL,0905           LXI  H,0905     文字列アドレス.
08FE   E5              PUSH   HL                PUSH H
08FF   CD B8 08        CALL   08B8 [ARPRINT]    CALL 08B8
0902   D1              POP    DE                POP  D
0903   C1              POP    BC                POP  B
0904   C9              RET                      RET

0905   6D697461726169 DB      'mitarai'         DB   'mitarai'
090C   00              DB     0                 DB   0
```

※オプションーe1000でコンパイル.

ところが，これをポインタを使って書き直してみると，リスト6-15のようにやや短くなります.

リスト 6-15　ポインタのテスト

```
arprint(pnt)
char    *pnt;
{
        while (*pnt)
                putchar (*pnt++);
}
main()
{
        arprint ("mitarai");
}
                        〈ザイログニー                  〈インテルニー
                           モニック〉                     モニック〉
```

```
ARPRINT:
08B8   C5              PUSH   BC                PUSH B
08B9   21 00 00        LD     HL,0000           LXI  H,0000
08BC   39              ADD    HL,SP             DAD  SP
08BD   F9              LD     SP,HL             SPHL            } フレームポインタの設定.
08BE   44              LD     B,H               MOV  B,H
08BF   4D              LD     C,L               MOV  C,L

08C0   CD 90 01        CALL   0190 [SDLI]       CALL 0190
08C3   04              DB     4                 DB   4
08C4   6E              LD     L,(HL)            MOV  L,M
08C5   7D              LD     A,L               MOV  A,L        } while (*pnt)
08C6   B7              OR     A,A               ORA  A
08C7   CA E2 08        JP     Z,08E2            JZ   08E2
```

```
08CA    21 04 00        LD      HL,0004                 LXI  H,0004   ┐
08CD    09              ADD     HL,BC                   DAD  B        │
08CE    5E              LD      E,(HL)                  MOV  E,M      │
08CF    23              INC     HL                      INX  H        │
08D0    56              LD      D,(HL)                  MOV  D,M      │
08D1    13              INC     DE                      INX  D        │
08D2    72              LD      (HL),D                  MOV  M,D      ├ *pnt++
08D3    2B              DEC     HL                      DCX  H        │
08D4    73              LD      (HL),E                  MOV  M,E      │
08D5    1B              DEC     DE                      DCX  D        │
08D6    EB              EX      DE,HL                   XCHG          │
08D7    6E              LD      L,(HL)                  MOV  L,M      │
08D8    26 00           LD      H,00                    MVI  H,00     ┘
08DA    E5              PUSH    HL                      PUSH H

08DB    CD F7 08        CALL    08F7 [PUTCHAR]          CALL 08F7     ┐
08DE    D1              POP     DE                      POP  D        ├ putchar( )
08DF    C3 C0 08        JP      08C0                    JMP  08C0     ┘

08E2    C1              POP     BC                      POP  B
08E3    C9              RET                             RET

MAIN:
08E4    C5              PUSH    BC                      PUSH B
08E5    21 EF 08        LD      HL,08EF                 LXI  H,08EF     文字列アドレス.
08E8    E5              PUSH    HL                      PUSH H
08E9    CD B8 08        CALL    08B8 [ARPRINT]          CALL 08B8
08EC    D1              POP     DE                      POP  D
08ED    C1              POP     BC                      POP  B
08EE    C9              RET                             RET

08EF    6D697461726169 DB       'mitarai'               DB   'mitarai'
08F6    00              DB      0                       DB   0
```

これはアルゴリズムの問題なので，必ずしも配列をポインタに置き換えられる訳ではありませんが，なるべくポインタを使うようにした方が有利です．

ただし，ポインタはC言語におけるスーパースターですが，あまり複雑な処理をコメントなしで記述してしまうと後で理解しにくくなりますので，注意してください．

なお，リスト6-14の引数ary，6-15の引数pntは本来まったく同じ形の設定なのですが，そのアクセス法が異なっています．6－15ではサブルーチンSDLIを用いた間接的な読み出しになっていますが，6－14ではじかにオブジ

ェクトが展開され，オプション－oを付けてコンパイルした時のようになっています．これは配列のアクセスではオフセットアドレスが固定のSDLIなどが使えないからです．これも配列を用いると不利になる点の一つです．

さて本節についてまとめておきます．

(a) コンパイル時には，－eオプションにより外部変数のアドレスを指定する．
(b) ローカル変数よりも外部変数を用いる．
(c) －oオプションはローカル変数，引数の参照を高速化するが，オブジェクトは大きくなる．
(d) －oオプションを使う時は，最も良く使うローカル変数（ループカウンタ，配列ポインタなど）を先頭で宣言する．
(e) 小さな関数はマクロ化する．
(f) ローカル変数に対しては，＋＋，＋＝などの高速演算子を使う．
(g) if文などではif（a！＝0）としても，if（a）としても全く同じなので，なるべく分りやすく記述する．
(h) if文などの条件式では一致しやすいステートメントを先に書く．
(i) 定数式は（　）で囲む．
(j) if文を羅列するより，switch文をつかう．
(k) switch文を使うときは評価する変数をなるべくchar型とし，一致しやすいcaseを先に書く．
(l) 配列を読み書きする場合はなるべくポインタをつかう．

6-3　ランタイムパッケージ

BDS Cのランタイムパッケージは作成したプログラムの中に必ず存在しているものです．この中に含まれているサブルーチンは標準関数と異なり，コンパイラが文脈を自動的に判断してCALLします．これらはBDS Cシス

テムの構造自体に関わる部分ですが，中にはアセンブラでプログラムを作成する際に利用できるものもあり，知っていると役立ちます．

ランタイムパッケージのソースファイルはCCC．ASMで，その中には，次のようなルーチンが含まれています．

(1) イニシャライズルーチン

プログラム実行の前処理を行ないます．スタックの設定や，CP／Mのコマンドラインを解析してmain関数に引数として渡したり，外部変数のクリアなどを行ないます．

これらをアセンブラルーチンなどから積極的に使う事はありません．しかし，ROM化を目的とするプログラムを作成したり，機能を拡張したい時にはランタイムパッケージを変更したい，あるいはしなければならない事があります．

まず，プログラムがスタートした後，処理は次のような順に行なわれます．（CP／Mの時）

スタックの設定はランタイムパッケージの先頭で行なわれます．この部分は，リンク時に書き換えられる部分で，リンク時のオプションによりプログラムそのものが変わります．オプションがなければ，

```
LD   HL,(6)     |   LHLD  6
LD   SP,HL      |   SPHL
```

となります．6番地にはBDOSの先頭アドレスが格納されていますから，BDOSのすぐ上にスタックを設定します．リンク時にスタック設定オプション－tを用いると直接スタックのアドレスを設定できるようになります．例えば，－t 8000というオプションが指定されると，

```
LD   SP ,8000H      |  LXI  SP, 8000H
NOP                 |  NOP
```

となります．また．－n(NOBOOT)オプションがあれば，

```
JP   SNOBSP         |  JMP   SNOBSP
NOP                 |  NOP
```

となり，スタックの設定はSNOBSP (set no boot stack)というルーチンに委ねられます．SNOBSPは123番地からのジャンプテーブルを介してVSNBSPにジャンプし，CCPを壊さないような場所にスタックを設定し，CCPへのリターンアドレスをSPSAVというワークエリアに格納します．

main関数終了後は109番地(FEXITV)にジャンプしてきますので，ここのジャンプアドレスもNOBRETに書き換えます．これで，プログラム終了時にはリブートせず，リターンアドレスをSPSAVから復活してCCPに戻ります．

それ以外の初期化ルーチンはサブルーチンINITにすべてまとめられています．

〈INIT〉

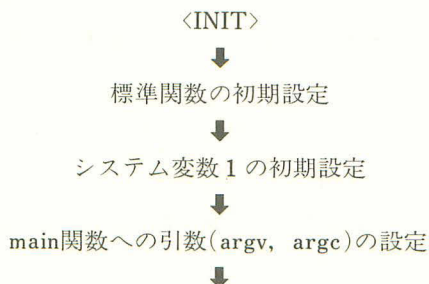
↓

標準関数の初期設定

↓

システム変数1の初期設定

↓

main関数への引数(argv, argc)の設定

ファイルインフォメーションテーブルのクリア

外部変数のクリア

システム変数2のクリア

リターン

最初は標準関数用の初期化ルーチンです．関数alloc，randのワークメモリの初期値を設定し，8080の命令ではできないinp, outp関数をコード書き換えで実現するためRAMへプログラムを書込んでいます．システム変数1とはCP／M上で走行する際のI／O関係の変数です．

次がmain関数への引数argv，argcの設定です．これはINITZZというラベル位置から始まりますが，コマンドラインの解析を行ない，ラベルCOMLIN（7F9H番地)から始まる131バイトにこの文字をコピーし，ラベルARGLST（87CH番地)からその文字列へのアドレスを列挙します．このポインタ配列によりmain関数はコマンドを受け取る事ができるのです．しかしポインタ用のメモリは60バイトしか確保されていないので，もしコマンド行に30個以上文字列を並べると，あっさり暴走します．

なお，コマンドラインに与える文字列はスペースがセパレータです．もし，文字列中にスペースを含ませたい場合は，その前後を“　”で囲むとスペースを含む文字列をmain関数で受け取る事ができます．ダブルコーテーションマーク(“　)そのものは除かれます(これは，Cの標準的な機能ではありません)

BDS　Cでは9つまでファイルをオープンするためランタイムライブラリ中にFCB(ファイルコントロールブロック)をとっています．このFCBテーブルがどういう状態にあるか示すフラグのテーブルがファイルインフォメーションテーブルで，27バイト(3バイト＊9個)確保されており，これをクリアします．

次は外部変数のクリアです．これは，単独にVCLREXというサブルーチン

になっていますが，一旦ランタイム先頭のCLREXというジャンプテーブルを介してCALLします．

```
CLREX：JMP　VCLREX
```

しかし，リンク時に－zオプションが与えられ外部変数をクリアしない時はここが，

```
CLREX：RET
```

に書き換えられ，VCLREXを実行しなくなります．

最後に標準関数ungetcで使うプッシュバックバイト(UNGETL)，errnoで使うエラーナンバ(ERRNUM)をクリアしてINITルーチンを終了します．

このINITの中にはCP／Mインタフェース用の処理が多いため，CPMというフラグによる条件アセンブルの指定が細かくなされています．ファイル先頭で

```
CPM　EQU　TRUE
```

となっているのをFALSEに変更すると殆どが削除されます（詳細は第7章を参照して下さい）．

良く"ランタイムパッケージには触らない事"と述べてある書籍を見掛けますが，実際にはソースファイルが公開されているうえ，コメントも完全で非常に解析しやすいので，むしろ自分のプログラムに合せてインストールする方が好ましいと思います．特にINITルーチンは最小公倍数的なプログラムですから不必要な部分を削除してみるだけでも結構プログラムサイズを小さくするのに役立ちます．

(2) 変数取り出しルーチン

ローカル変数，外部変数の内容を取り出すルーチンで，SDLI，LDLI，SDEI，LDEIなどがこれに当たります．

このルーチンは変数アクセスの基本になる重要な部分ですから，アドレスは固定されています．これらの内容を変更したいときは，エントリアドレスが変わらないように注意しなければなりません．

このルーチンのエントリアドレス，機能を表にしておきます．

アドレス	ラベル	機　　能	取り出す値	オフセットビット数
014D	LDEI	外部変数参照	16ビット	16ビット
015C	SDEI	外部変数参照	16ビット	8ビット
016B	LSEI	外部変数参照	8ビット	16ビット
0177	SSEI	外部変数参照	8ビット	8ビット
0183	LDLI	ローカル変数参照	16ビット	16ビット
0190	SDLI	ローカル変数参照	16ビット	8ビット

オフセットのビット数が16ビットのものは，

```
CALL    LxyI
DW      nn
```

8ビットのものは，

```
CALL    SxyI
DB      n
```

のフォーマットで呼び出します．これで，HL,あるいはLレジスタに目的の変数の値が得られます．なお，n，nnは宣言の頭からのバイト数を指定します．

なお，外部変数を参照するLDEI，SDEI，LSEI，SSEIはコンパイル時に－eオプションを指定すればオブジェクト中からは呼ばれなくなりますので削除する事も可能です．ただし，標準関数の中にも外部変数を参照してい

るものがあるので，注意してください．LDLI，SDLIも－oオプションを付けてコンパイルすると使われなくなります．

(3) 引数取り出しルーチン

引数の参照は一般にローカル変数参照サブルーチンLDLI，SDLIを利用しますが，アセンブラ定義関数から使うのはかなり厄介なので専用に次のようなサブルーチンが用意されます．

MA 1TOH:	1番めの引数を取り出す	
MA 2TOH:	2	〃
MA 3TOH :	3	〃
MA 4TOH:	4	〃
MA 5TOH:	5	〃
MA 6TOH:	6	〃
MA 7TOH:	7	〃

これらは，呼び出された状態(BCレジスタをPUSHする前)に，

```
CALL    MAxTOH
```

とする事により，目的の変数の値がHLレジスタ，下位8ビットはAレジスタにも格納されます．ただし，注意しなければならないのはスタックの状態です．MAxTOHではスタックポインタの内容から引数のアドレスを求めますので，呼び出す前にPUSHなどスタックを操作する命令を用いると，順番が狂ってきます．

MAxTOHを呼び出す前に実行したPUSH命令の数だけ，

MA1TOH → MA2TOH

のように1つずつずらしてください．なお，MAxTOHはアセンブラ定義の標準関数を用いず，自分でも利用しない場合には不要です．

次に，このMAxTOHも利用するのが面倒な場合のため，ARGHAKというサ

ブルーチンも用意されています．このARGHAKは関数先頭でCALLすると，引数７つをARGSというデータエリア内にコピーします．以降，どのように複雑な処理をしている最中でも，

　　LHLD　ARGn　（nは1から7）

とするだけで引数を参照できます．ただし，注意しなければならないのは一旦固定したアドレスにデータを格納するため，リエントラントな性質が失われることです．つまり，ARGHAKを用いている関数からさらにARGHAKを用いる関数を呼び出す事はできないのです．一番簡単に引数を参照するルーチンですが，アセンブラで関数を定義する際，他の標準関数を呼び出したり，インタラプトを用いる場合に，この落し穴には気を付ける必要があります．

(4)　演算ルーチン

8080にない，乗算，除算などのルーチンです．

SMUL :　符号付き乗算（HL＝DE＊HL）
USMUL :　符号なし乗算（HL＝DE＊HL）
SDIV :　符号つき除算（HL＝DE／HL）
USDIV :　符合なし除算（HL＝DE／HL）
SMOD :　符号つきMOD演算（HL＝DE％HL）
USMOD :　符号なしMOD演算（HL＝DE％HL）

これらのサブルーチンはすべて8080の命令で作成されていますので，Z80を用いる場合にはもう少し小さく，高速になります．力のある方は試みてください．

なお，上記の他にシフト，比較など小さいサブルーチンがたくさんあります．これらは簡単なものですから必要な方は解析してください．なお，これらのサブルーチンをプログラム中で利用したい時はBDS．LIBをインクルードすると，ラベルで参照できます．

CCC．ASMは判りやすいすぐれたプログラムです．初心者の方がアセンブ

ラを勉強する際には極めて良質なサンプルになると思います.

6-4 BDS Cのバグ

BDS Cは問題の少ないソフトウェアですが，残念ながらいくつかバグがあります．この他にもあると思いますが大きなものを紹介しておきます．

＊＃defineで無限に参照を繰り返すようなマクロ定義をすると暴走します．

```
#define  ABC  DEF
#define  DEF  ABC
main ( )
{
        int  a
        a=ABC;
}
```

上記のようなプログラムはマクロ検索が無限に続くため，CC. COM(パス1）の処理が終了しません(暴走します)．

本来，このようなマクロ定義をしてはいけないのですが，プログラムが長くなり，変更を繰り返していたりするとついうっかりこのようなパターンになってしまうことがあります．いつまでたってもコンパイルが終らないので時間も無駄になりますし，リセットするか電源を入れ直さなければならないので手間もかかります．

エラーメッセージを表示して終了するように改善して欲しいものです．

このバグはBDS C，α-C両方とも同じです．

第 7 章
ROM化の方法

BDS　Cコンパイラを用いると，CP/M以外の環境で実行できるプログラムを作成する事ができます．

この機能により制御用のボードコンピュータや，汎用OSを持たない専用のシステムのためのプログラム開発が効率的に行なえます．このようなソフトは一般にROMに格納されてシステムに組み込まれる場合が多いため，ROM化プログラムと呼ばれて区別されます．今まで，この用途はアセンブラ言語の独壇場でしたが，C言語を用いると開発期間が短くなる，移植性が高い，変更修正が容易で保守性も良い，などその効果が大きいため，最近はかなり頻繁に行なわれるようになってきました．

ただし，注意しなければならないのは，C言語を用いてもソフト開発そのものは必ずしもアセンブラより便利になるとは言い切れないという事です．これは，異論を持たれる方も少なからずいるとは思いますが，Cは基本的にOS上で動作するプログラムを作成する事を前提に考えられています．従って特殊な環境で走行させようとすると，C言語としての利点が生きず，意外と作成が面倒になるものなのです．

BDS　CもあくまでCP/M上でのプログラム作成に重点をおいたコンパイラですから，「アセンブラができないからBDS　Cで」という安易な発想では，まず間違いなく途中で挫折する事になるでしょう．標準関数は殆ど使えなくなりますし，アセンブラで作成しなければならない部分も出てきます．また，CDBが使えませんからその場合において自分自身でデバッグ法を考えていく必要があります．これらの注意はROM用ソフト作成の経験がある方には自明の事です．

しかし，ある程度慣れれば一般にCP/M上でROM用プログラムを開発する際には，確実にアセンブラよりも早くできます．特に大きなプログラムほどその効果は顕著に現れるでしょう．

ROM化する場合には，知らなければならない事としなければならない作業が数多くあります．即効を期待せず，数あるROMプログラム作成法の一つとして十分に吟味検討される事をおすすめします．

7-1 ROM化の条件

BDS CでROM用ソフトウェアを作る際にはいくつかの制約があります．プログラムが殆どC言語で記述できるのか，かなりの部分をアセンブラで作成しなければならないのかなど，開発を始める前にある程度検討しておく事が必要です．

ROM化だけではなく，他のOS上で動作させるプログラムを作成する場合にも殆ど同じ事が言えます．この場合プログラムはRAM上で動作する事になりますので，若干異なる部分もありますが，基本的な注意点はROM化の際と同じです．

まず，ROMソフトの場合，ハードウェアによってBDS Cが使えない事があります．それは，次のような時です．

(1) プログラムエリアが極端に少ない場合

BDS Cには，最低限必要なランタイムライブラリ（約700バイト）があり，これよりプログラムサイズを小さくする事はできません（勿論，厳密には可能です．第6章を参照してください）．また，プログラムのバイト数の管理を厳密にする事が難しく，ローカル変数を利用する際などかなりプログラムが大きくなりますので，メモリに余裕がない時は注意する必要があります．プログラムの内容によりますが，アセンブラで作成する場合に比べ，2倍程度のプログラムメモリを用意する必要があると考えて良いでしょう．

(2) RAMエリアが非常に小さい場合

ランタイムライブラリだけでRAMを110バイト消費します（ライブラリの変更により，減らす事は可能です）．また，ローカル変数と関数への引数をスタック上に設定しますので，スタック用のRAMも多めに用意しておく必要があります．

(3) 高速な処理を必要とする場合

制御用のソフトではリアルタイムに近い高速な処理が要求される事があります．程度によりますが，アセンブラでも技巧を必要とするような内容のものは難しいと考えた方が良いでしょう．ただし，部分的に高速な処理が必要といった程度であれば，そこだけアセンブラで記述すれば，作業を合理化する事ができます．

(4) 特殊なメモリ構成をとるハードウェアの場合

メモリの拡張のためにバンク切り換えを用いたり，特殊なメモリマネージメントユニット（MMU）を使用したりする時にはその構成にもよりますが，かなりプログラム作成が難しくなります．

(5) インタラプトを多用するプログラムの場合

BDS Cは基本的にインタラプトをサポートしていません．パッチにより，インタラプト処理部分をリンクしなければならないため，手続きが面倒です．

また，α-CではROM化に必要なファイルが含まれていませんので，ROM用プログラムの作成はできません．

次に，BDS Cでプログラムを作成するのが有利だと思われるのは，次のような場合です．

(1) プログラムがかなり大規模な場合

アセンブラで作成してもかなり大きなプログラムになる場合には，Cで作成すると開発期間が短くなります．ただし，8080，Z80ではもともと64Kバイトのアドレス空間しかありませんからアセンブラでも20-30Kバイトにもなるプログラムであれば，C言語で作成すると物理的にプログラムがROMに入らなくなる可能性もあります．

ただ，一般に30Kバイト以上のアセンブラプログラムというのはCP/Mにおけるツール自体の能力としても問題のでてくる容量ですから，いずれにし

てもその開発の方法そのものを十分検討する必要があります．

(2) リエントラントなプログラムを作成する場合

前項の(5)と相反するようですが，インタラプトを用いたり，マルチタスク機能を持つシステムで，別々のタスクから呼び出せる共通なサブルーチンを作成したい場合にBCS Cは便利です．BDS Cでは，ローカル変数を用いるとプログラムが呼び出されるたびに変数領域を確保しますので，どのような状態で呼び出されても，他の変数を破壊したりする事がなく，正しいプログラムの実行が保たれるからです（リエントラント）．

8080，Z80ではローカル変数を持たせる事は難しく，アセンブラによるリエントラントなプログラムの作成はかなりな負担になりますから，手間がかなり省けます．Z80のIX，IYレジスタは使用しませんのでプログラムの可読性はやや落ちますが，オブジェクトのサイズやスピードには余り影響しないと思います．

大体以上のような事を念頭に置いておくと作成を検討する場合に参考になるでしょう．

なお，インタラプトを使用するシステムにおいてBDS Cを使う場合，一般にコンパイルやデバッグ作業は不便になりますが，プログラムの作成は簡単になります．これはBDS Cというより，C言語そのものの問題なのですが，C言語では基本的に複数のプログラムが同時に走行するような記述ができません（勿論BASICにもPascalにもそのような機能はありません）．しかし，その大前提となるリエンラントなコード展開は確保されていますので，いざ本格的にインタラプトを使ったプログラムを作成する場合には強力なツールになるのです．

ただ，短いプログラムであれば，インタラプト部分をすべてアセンブラで記述する方が簡単でしょう．

7－2　ROM化の手順

ROM化するプログラムを作成する場合には，BDS Cシステムを ROM 用に改造しなければなりません．その手順はかなり面倒ですが，重要な作業ですのでここで詳細に述べておきます．

サンプルとしては，ROM上ではなく，他のOSで動作するプログラムを作成してみました．LSIのイニシャライズなどはする必要がありません．

ハードウェア：	SHARP X1ターボ
動作環境：	BASIC
プログラム領域：	E000番地—EFFF番地（4Kバイト）
変数およびスタック領域：	F000番地—F7FF番地（2Kバイト）
プログラム内容：	グラフィック画面を1／2サイズへ縮小

プログラム領域は，E000番地からですから，このアドレスでプログラムが動作するようにランタイムパッケージを変更します．

一般にROMプログラムは0番地から配置する事が多いので，その場合には，ランタイムパッケージC.CCCは100Hから配置するようにしておくとプログラムをCP／M上でデバッグしやすくなります．特に，割り込みがあるシステムではリスタート命令を用いて0-38H番地にジャンプさせる場合が多いのでこのエリアをBDS Cのライブラリルーチンに占領されてしまうと割り込みを使いずらくなります．

メモリの配置が決まったら，システムを改造します．

CCC.ASMはBDS Cのランタイムパッケージ C.CCCのソースファイルです．オブジェクトがCP／M以外の環境で動作するように変更します．

リスト7－1　CCC.ASMの変更

```
24: FALSE:  EQU     0
25: TRUE:   EQU     NOT FALSE
26:
27: CPM:    EQU     TRUE
```

```
28:
29: MPM2:   EQU     FALSE
30:
31: DMAVIO: EQU     TRUE
```

⇓

```
24: FALSE:  EQU     0
25: TRUE:   EQU     NOT FALSE
26:
27: CPM:    EQU     FALSE
28:
29: MPM2:   EQU     FALSE
30:
31: DMAVIO: EQU     FALSE
```

FALSEに変更する.

```
54:             IF NOT CPM
55: ORIGIN: EQU     NEWBASE
56: RAM:    EQU     WHATEVER
57:
58: EXITAD: EQU     WHENDONE
59:             ENDIF
```

⇓

```
54:             IF NOT CPM
55: ORIGIN: EQU     0E000H
56: RAM:    EQU     0F000H
57:
58: EXITAD: EQU     0000H
59:             ENDIF
```

ROMのアドレスを設定する.

RAMのアドレスを設定する.

EXITADはプログラム終了時にジャンプするアドレス. main関数終了時と関数exitで使用するが，サンプルでは特に用いないので0番地に設定.

```
127:            IF      NOT CPM
128:            jmp     verror
129:            jmp     verror
130:            jmp     verror
131:            jmp     verror
132:            jmp     verror
133:            jmp     verror
134:            jmp     verror
135:            jmp     verror
136:            jmp     verror
137:            jmp     verror
```

```
138:          jmp    verror
139:          ENDIF
```

↓

```
127:          IF     NOT CPM
128:          jmp    verror
129:          jmp    verror
130:          jmp    verror
131:          jmp    verror
132:          jmp    verror
133:          jmp    verror
134:          jmp    verror
135:          jmp    verror
136:          jmp    verror
137:          jmp    verror
138: clrex:   jmp    verror
139:          ENDIF
```

↑ ラベルを追加する．

〈C.CCC の再アセンブル〉

```
A>ASM CCC
CP/M ASSEMBLER - VER 2.0
F06F ←          RAMの先頭アドレスから6EH(110)バイト
                消費しており，利用可能なのはF06FH番地から．
006H USE FACTOR
END OF ASSEMBLY

A>DDT
DDT VERS 2.2

-H100 E000 ←    100H－E000Hを計算する．
E100 2100       2100Hが求める値．

-ICCC.HEX
-R2100 ←        E000H番地からはじまるCCC.HEXを
                100番地からロード．

NEXT  PC
03B2 0000
-^C

A>SAVE 3 C.CCC ← ここで「3」はCCC.HEXをロードした時の
A>               NEXT　03B2の上位「03」である．
```

リスト7-1を参考に，CCC.ASMの27，31，55，56，58，138行付近の6箇所を変更します．この中で，58行めのEXITADはプログラムの終了時にジャンプするアドレスですが，無限ループなどで終了する事がないプログラムであれば，適当に決めます．今回のようにBASIC上で動作させる場合は，ホットスタートや，エラートラップのようにスタックを回復してからBASICに戻る番地を指定しておきます．

また138行めはマニュアルにはありませんが，clrexというラベルを付けておかないと後でアセンブルエラーになります(これは恐らくBDS Cの作者ゾールマン氏が付け忘れたのでしょう)．

CCC.ASMを修正したら，それをASMでアセンブルします．これでCCC.HEXファイルができますから，DDTを起動しRコマンドで100H番地からロードします．DDTではこの場合のように100Hから始まらないHEXファイルを100Hからロードするには，デフォルト値100Hからプログラムの先頭番地を引いた値をRコマンドのオフセットアドレスとします．この時の計算にはHコマンドを使うと便利です（Hコマンドは16進で2つの値の和と差を計算します)．

この作業が終了したら，今度はそれに対応するようにBDS.LIBを修正します．BDS.LIBはアセンブラ定義関数にランタイムパッケージのエントリアドレスを与えるファイルで，修正点も大体CCC.ASMと同じです．

リスト7-2 BDS.LIBの変更

```
16: CPM:    EQU 1
17: MPM2:   EQU 0
18: ;
19: ; System addresses:
20: ;
21:
22:         if not cpm
23: CCCORG: EQU WHATEVER
24: RAM:    EQU WHATEVER2
25: BASE:   EQU WHATEVER3

       ↓↓
```

```
16: CPM:    EQU 0
17: MPM2:   EQU 0
18: ;
19: ; System addresses:
20: ;
21:
22:         if not cpm
23: CCCORG: EQU 0E000H<──────── ROMアドレス
24: RAM:    EQU 0F000H<──────── RAMアドレス
25: BASE:   EQU 0000H<───────── EXITADのアドレス
                                (リスト7−1参照)
```

さて，次にCP／M以外の環境でも利用できる標準関数のうち，アセンブラで定義されているものを整理します．この作業は，まず使える関数をより分ける事から始めます．

アセンブラ定義標準関数のファイルは，DEFF2A. CSM，DEFF2B. CSM，DEFF2C. CSM，DEFF2D. CSMの4つで，BDS. LIBはアセンブル時にインクルードされます．この中で利用可能な関数には次のようなものがあります．

DEFF2A.CSM	rand, setmem, movmem, call, calla, inp, outp, peek, poke, sleep, exit, codend, externs, endext
DEFF2B.CSM	index, setjmp, longjmp
DEFF2C.CSM	なし
DEFF2D.CSM	fp, long

ソースファイルからいらない関数の定義部分を削除しても良いのですが，かなり面倒な作業になりますので，エラーがでるのを覚悟でそのままアセンブルし，CRLファイルを作成してから必要な関数をピックアップした方が簡単でしょう．

この方法だと，ROMプログラムのアドレスが100Hから始まるときはアセンブルする必要もなくなります．作業後のファイル名は，DEFF2A. CRL，DEFF2B. CRL，DEFF2D. CRLとします．

操作例7-3　アセンブラ定義関数の再アセンブル

```
A>SUBMIT CASM DEFF2A

A>XSUB
A>CASM DEFF2A

BD Software CRL-format ASM Preprocessor v1.50
Processing the GETCHAR function...
Processing the KBHIT function...
                    ¦（中略）
Processing the RSVSTK function...
Processing the MEMCMP function...
DEFF2A.ASM is ready to be assembled.

(xsub active)
A>ASM DEFF2A.AAZ

CP/M ASSEMBLER - VER 2.0
U0318 0E00             mvi    c,conin  ⎫
U031A CD0000           call   bdos     ⎪
                    ¦（中略）          ⎬ BDOSコールなどを使う関数はすべてエラーがでるが，
U0754 CD0000           call   bdos     ⎪ ここではどうせ使わないので無視する．
U0778 CD0000           call   bdos     ⎪
U07F8 3A0000           lda    tpa      ⎭
S0008 =          SECTORS$ EQU ($-TPALOC)/256+1 ;USE FOR "SAVE" !.
01F4
01DH USE FACTOR
END OF ASSEMBLY

(xsub active)
A>DDT DEFF2A.HEX
DDT VERS 2.2
NEXT  PC
08F1 0000
-G0

(xsub active)
A>ERA DEFF2A.ASM
A>ERA DEFF2A.HEX

A>SAVE 8 DEFF2A.CRL
```

〈DEFF2Bのアセンブル〉

```
A>SUBMIT CASM DEFF2B

A>XSUB
A>CASM DEFF2B

BD Software CRL-format ASM Preprocessor v1.50
Processing the INDEX function...
                 ¦
                 ¦
Processing the TXTPLOT function...
DEFF2B.ASM is ready to be assembled.

(xsub active)
A>ASM DEFF2B.AAZ

CP/M ASSEMBLER - VER 2.0

U0362 0E00              mvi     c,getlin
U0365 CD0000            call    bdos
U0368 0E00              mvi     c,conout
U036C CD0000            call    bdos
S0005 =         SECTORS$ EQU ($-TPALOC)/256+1 ;USE FOR "SAVE" !.
014B
000H USE FACTOR
END OF ASSEMBLY

(xsub active)
A>DDT DEFF2B.HEX
DDT VERS 2.2
NEXT  PC
0548 0000
-G0

(xsub active)
A>ERA DEFF2B.ASM
A>ERA DEFF2B.HEX

A>SAVE 5 DEFF2B.CRL
```

〈DEFF2Dのアセンブル〉

```
A>SUBMIT CASM DEFF2D

A>XSUB
A>CASM DEFF2D
```

```
BD Software CRL-format ASM Preprocessor v1.50
Processing the FP function...
Processing the LONG function...
DEFF2D.ASM is ready to be assembled.

(xsub active)
A>ASM DEFF2D.AAZ
CP/M ASSEMBLER - VER 2.0
S000C =         SECTORS$ EQU ($-TPALOC)/256+1 ;USE FOR "SAVE" !.
010D
026H USE FACTOR
END OF ASSEMBLY

(xsub active)
A>DDT DEFF2D.HEX
DDT VERS 2.2
NEXT  PC
0C9F 0000
-G0

(xsub active)
A>ERA DEFF2D.ASM
A>ERA DEFF2D.HEX

A>SAVE 12 DEFF2D.CRL
A>
```

次に，標準関数のうちCで作成されているものを再コンパイルします．ソースファイルは，STDLIB1.CとSTDLIB2.Cの2つですが，利用できるものは次のとおりです．

```
STDLIB1.C    atoi,strcat,strcmp,strcpy,
             strlen,isalpha,isupper
             islower,isdigit,isspace,
             toupper,tolower,qsort
             initw,initb,getval,abs,max,min
STDLIB2.C    sprintf,_spr,sscanf,_scn
```

これらについては，特別な書き換えは必要ありませんが，ランタイムライブラリーのアドレスがE000番地になっているため，－mオプションをつけてコンパイルします．

操作例 7－4　C定義関数の再コンパイル

```
A>cc stdlib1.c -me000
```

Cで定義された標準関数を再コンパイルする．この時C.CCCが0E000Hから始まっている事を指定する．

```
BD Software C Compiler v1.50a (part I)
  27K elbowroom
BD Software C Compiler v1.50 (part II)
  25K to spare


A>cc stdlib2.c -me000

BD Software C Compiler v1.50a (part I)
  29K unused
BD Software C Compiler v1.50 (part II)
  27K to spare
```

さて，これで一応必要な関数の変換作業が終りました．最後に，一番重要なCRLファイルの整理を行ないます．実際は使用する関数だけをピックアップしておけば十分ですが，ここでは使用可能な関数をすべて一つにまとめたライブラリファイルを作ってみます．

CRLファイルの整理には，CLIB. COMを使います．CLIBはCRLライブラリーファイルを作成する時に使うプログラムで，次のようなコマンドがあります．

リスト7－5　CLIBのコマンド

```
A>clib

BD Software C Librarian  v1.50
Function buffer size = 44222 bytes.

*help
C Librarian commands:
   (note: "file#" is a single digit, 0-9)

o[pen]     file# [d:]filename
f[iles]
e[xtract]  file# funcname
a[ppend]   file# [funcname]
d[elete]   file# funcname
```

```
r[ename]   file# funcname_old funcname_new
c[lose]    file#
q[uit]     [file#]
m[ake]     filename
t[ransfer] source_file# dest_file# funcname
h[elp]
l[ist]     file#
```

このCLIBのコマンドを使って，次のように関数を転送します．

操作例7-6 ライブラリ関数の整理

```
A>CLIB
BD Software C Librarian  v1.50
Function buffer size = 41662 bytes.

*MAKE DEFF ←—— DEFF.CRLというファイルを作成する．
*OPEN 1 DEFF ←—— DEFF.CRLをオープンし書込む用意をする(ファイル1)．
*OPEN 0 STDLIB1 ←—— STDLIB1.CRLをオープンし，読出しにそなえる(ファイル0)．

*FILES —— オープンしたファイルの確認．
file #0 (A:STDLIB1.CRL) size= 5573 bytes; dir space= 275/512 bytes
file #1 (A:DEFF.CRL) size= 516 bytes; dir space= 511/512 bytes

*T 0 1 ATOI ←—— ファイル0からファイル1にATOI関数を転送．
*T 0 1 STRCAT
*T 0 1 STRCMP
*T 0 1 STRCPY
*T 0 1 STRLEN
*T 0 1 ISALPHA
*T 0 1 ISUPPER
*T 0 1 ISLOWER
*T 0 1 ISDIGIT
*T 0 1 ISSPACE
*T 0 1 TOUPPER
*T 0 1 TOLOWER
*T 0 1 QSORT
*T 0 1 _SWP ←—— _SWPはQSORT中で使われる関数．
*T 0 1 INITW
*T 0 1 INITB
*T 0 1 GETVAL
*T 0 1 ABS
*T 0 1 MAX
*T 0 1 MIN
*CLOSE 0 ←—— 必要な関数の転送が終了したため，DEFF2A.CRLをクローズ．
Please wait while this crunches away...
```

```
*OPEN 0 STDLIB2
*T 0 1 SPRINTF
*T 0 1 _SPR   )
*T 0 1 _SSPR  | この4つの関数は
*T 0 1 _USPR  | SPRINTFで使われる.
*T 0 1 _GV2   )
*T 0 1 SSCANF
*T 0 1 _SCN   )
*T 0 1 _IGS   | SSCANF中で使用される関数.
*T 0 1 _BC    )
*CLOSE 0
Please wait while this crunches away...

*OPEN 0 DEFF2A
*T 0 1 RAND
*T 0 1 SETMEM
*T 0 1 MOVMEM
*T 0 1 CALL
*T 0 1 CALLA
*T 0 1 INP
*T 0 1 OUTP
*T 0 1 PEEK
*T 0 1 POKE
*T 0 1 SLEEP
*T 0 1 PAUSE
*T 0 1 EXIT
*T 0 1 CODEND
*T 0 1 EXTERNS
*T 0 1 ENDEXT
*CLOSE 0
Please wait while this crunches away...

*OPEN 0 DEFF2B
*T 0 1 INDEX
*T 0 1 SETJMP
*T 0 1 LONGJMP
*CLOSE 0
Please wait while this crunches away...

*OPEN 0 DEFF2D
*T 0 1 FP
*T 0 1 LONG
*CLOSE 0
Please wait while this crunches away...
```

```
*FILES
file #1 (A:DEFF.CRL) size= 8448 bytes; dir space= 163/512 bytes

*LIST 1◄──────────────ファイル1 (DEFF.CRL)に含まれる
                        関数を確認する.

A:DEFF.CRL:
ATOI: 222  STRCAT: 111  STRCMP: 158  STRCPY: 83
STRLEN: 72  ISALPHA: 85  ISUPPER: 54  ISLOWER: 54
ISDIGIT: 54  ISSPACE: 60  TOUPPER: 78  TOLOWER: 78
QSORT: 384  _SWP: 108  INITW: 93  INITB: 89
GETVAL: 131  ABS: 44  MAX: 48  MIN: 48
SPRINTF: 65  _SPR: 1165  _SSPR: 35  _USPR: 194
_GV2: 121  SSCANF: 46  _SCN: 926  _IGS: 92
_BC: 196  RAND: 49  SETMEM: 38  MOVMEM: 96
CALL: 37  CALLA: 40  INP: 18  OUTP: 18
PEEK: 12  POKE: 16  SLEEP: 61  PAUSE: 19
EXIT: 8  CODEND: 9  EXTERNS: 9  ENDEXT: 9
INDEX: 69  SETJMP: 31  LONGJMP: 41  FP: 1772
LONG: 686

*CLOSE 1
Please wait while this crunches away...

*QUIT◄──────────────CLIB の終了.
(Quit) Are you sure (y/n)? Y

A>
```

操作例7-6でわかるように，stdlib1，stdlib2中の関数を先に転送するのが原則です．Cで定義されている関数はアセンブラ定義関数を利用している場合がありますから，このようにしておかないと正しくリンクできない可能性があります．（操作例7-6の場合は多分問題ないでしょう）このリンクのアルゴリズムについては2-4節を参照してください．

これで，E000番地バージョンの標準関数ライブラリファイルDEFF. CRLが完成します．変更したDEFF. CRL，C. CCCの2つはファイル名がCP／M用の標準ファイルと同一なので，混同しないよう新しいディスクに移しておきます．この時最低限必要なBDS Cのシステムファイルは次のようになります．

CC.COM
CC2.COM
C.CCC　　　(変更したもの)
DEFF.CRL (変更したもの)
CLINK.COM (あるいはL2.COM)

このディスクに，必要に応じてユーティリティプログラム（CASM，アセンブラなど）をコピーしておくのは，言うまでもない事です．

7-3 プログラムの開発

プログラムの開発に当ってまず考えなければならない事は，プログラムのデバッグです．そのままやみくもにプログラムを書き，実行しようとしても，必ず行き詰まります．ROMの場合は特にPROMへの書込み，消去作業に時間がかかりますから，一層の注意を必要とします．ましてや，完全に動作する事が確認されていないようなハードウェアでは，デバッグの方針を十分に立てておくべきです．

プログラムをターゲットシステムで動作させる前にCP／M上で可能な限り関数単位のデバッグをしておく方法も良く用いられます．

まず，サンプルプログラムの概要を解説します．

今回のプログラムは，ROM化ではなくBASICで作成したプログラムを高速化する目的でCを使った例です．BDS Cの使い方としては，かなり珍しい部類にはいると思いますが，ROM化ではそれぞれの場合で全くデバッグ法が異なるうえ，その例を実際に読者の方がテストしようと思ってもまずできませんから，あまり参考になりません．むしろ，BASICからBDS Cで作成されたルーチンを呼び出すという例の方が参考になるでしょう．

さて，ここではグラフィック画面の縮小プログラムを作成してみます．この最も簡単なアルゴリズムはピクセル（画素）を1つ飛びにピックアップし

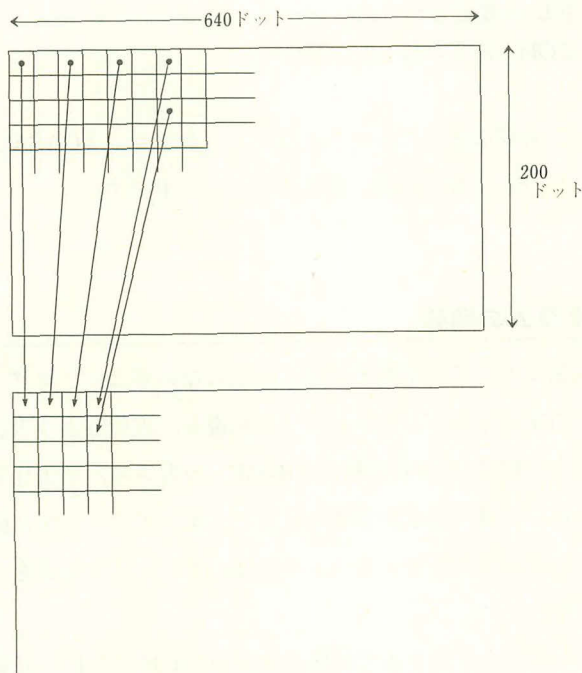
ていくもので，さらに完成画面の設定を工夫すると画面の退避の問題もなくなりますから，非常に簡単です．

（図のように画面上のドットを1つおきにコピーし，½に画面縮少する．）

図7-7 画面の縮少

このアルゴリズムに従って，プログラムを作成します．この時，CP／M上でデバッグする事を前提に，CP／M上で動作する部分と，動作しない部分に分け，後者をシミュレートするテストプログラムを作成します．

一般にはこの両者を完全に2つのファイルに分けておきますが，ここでは，ハードに依存する部分とそうでない部分で2つに分け，reduce. c とreduce2. cとしました．reduceがメインプログラムでそのまま最終的に用いる部分，reduce2はあとで一部をアセンブラで置き換える部分です．また，両者に共

通なヘッダーファイルをreduce.hとします．

プログラムと，そのデバッグ経過を操作例7-8に示します．

操作例7-8 reduceのCP／M上でのデバッグ

A>type reduce.h

```
int      x,y;     /* location      */
int      ty;
char     bitmask;
char     r,g,b;
unsigned         adr;

#define DEBUG   1
```

<reduce.h>
(外部変数の設定を別ファイルにしておく)

A>type reduce.c

```
#include        <reduce.h>

#define XSIZE   640/2
#define YSIZE   200/2

reduce()
{
        for ( y = 0; y < YSIZE; y++ )
        {
                ty = y * 2;
                for ( x = 0; x < XSIZE; x++ )
                {
#if DEBUG
                        printf ("¥n¥nx=%3d y=%3d",x,y);
#endif
                        pset ( x, y, point ( 2 * x, ty ));
                }
        }
}
```

<reduce.c>
メインプログラム
(CP/M上でデバッグ時とターゲット走行時に変更しない部分をまとめておく)

A>type reduce2.c

```
#include        <reduce.h>

pset(x,y,c)
int     x,y;
char    c;
{
        bitmask = decode ( x & 7 );
#if DEBUG
        printf ("¥npset function bit= %d¥n",bitmask);
```

<reduce2.c>
(デバッグ用関数が加えられているファイル)

```
#endif
        g = ((c & 4)==0) ? 0: bitmask;
        r = ((c & 2)==0) ? 0: bitmask;
        b = ((c & 1)==0) ? 0: bitmask;

        adr = ((y / 8 * 80) + (y & 7) * 0x800) + x/8 + 0x4000;

        outbc (adr, ((~bitmask) & inbc (adr)) | b);
        adr += 0x4000;
        outbc (adr, ((~bitmask) & inbc (adr)) | r);
        adr += 0x4000;
        outbc (adr, ((~bitmask) & inbc (adr)) | g);
}

char decode(a)
char    a;
{
        char    d;

        d = 128;
        while ( a>0 )
        {
                d /= 2;
                a--;
        }
        return (d);
}

point(x,y)
int     x,y;
{
        bitmask = decode ( x & 7 );
#if DEBUG
        printf ("¥npoint function bit= %d¥n",bitmask);
#endif
        adr = ((y / 8 * 80) + (y & 7) * 0x800) + x/8 + 0x4000;

        b = (inbc (adr) & bitmask) / bitmask;
        r = (inbc (adr + 0x4000) & bitmask) / bitmask * 2;
        g = (inbc (adr + 0x8000) & bitmask) / bitmask * 4;

        return (g+r+b);
}


#if DEBUG

main()
{
        reduce();
}
```

```
inbc(adr)
unsigned        adr;
{
        printf ("inbc adr= %x  ",adr);
        return(0);
}

outbc(adr,c)
unsigned        adr;
char    c;
{
        printf ("outbc adr= %x  ",adr);
}

#endif
```

```
A>cc reduce     ←── CP/Mデバッグ用プログラムのコンパイル
BD Software C Compiler v1.50a (part I)
  35K elbowroom
BD Software C Compiler v1.50 (part II)
  32K to spare

A>cc reduce2
BD Software C Compiler v1.50a (part I)
  34K elbowroom
BD Software C Compiler v1.50 (part II)
  31K to spare

A>l2 reduce reduce2     ←── CLINKでも良いが，
                            CLINKの場合にはリンクファイルの
                            最初のファイルにmain関数が含まれ
                            ていなければならないので，この場
                            合は不便．
Mark of the Unicorn Linker ver. 2.2.2
Loading  REDUCE.CRL
Loading  REDUCE2.CRL
Scanning DEFF.CRL
Scanning DEFF2.CRL


Link statistics:
  Number of functions: 15
  Code ends at: 0x12E3
  Externals begin at: 0x12E3
  Externals end at: 0x12EF
  End of current TPA: 0xDC06
  Jump table bytes saved: 0x63
  Link space remaining: 23K

A>reduce     ←── デバッグ実行
                 (xが正しくインクリメントされ，正しくアドレス)
                 (計算がなされている事をチェックすればよい    )
```

```
x=  0 y=  0
point function bit= 128
inbc adr= 4000  inbc adr= 8000  inbc adr= C000
pset function bit= 128
inbc adr= 4000  outbc adr= 4000  inbc adr= 8000  outbc adr= 8000  inbc adr= C000  outbc adr= C000

x=  1 y=  0
point function bit= 32
inbc adr= 4000  inbc adr= 8000  inbc adr= C000
pset function bit= 64
inbc adr= 4000  outbc adr= 4000  inbc adr= 8000  outbc adr= 8000  inbc adr= C000  outbc adr= C000

x=  2 y=  0
point function bit= 8
inbc adr= 4000  inbc adr= 8000  inbc adr= C000
pset function bit= 32
inbc adr= 4000  outbc adr= 4000  inbc adr= 8000  outbc adr= 8000  inbc adr= C000  outbc adr= C000

x=  3 y=  0
point function bit= 2
inbc adr= 4000  inbc adr= 8000  inbc adr= C000
pset function bit= 16
inbc adr= 4000  outbc adr= 4000  inbc adr= 8000  outbc adr= 8000  inbc adr= C000  outbc adr= C000
```

（中略）

```
x=318 y=  0
point function bit= 8
inbc adr= 404F  inbc adr= 804F  inbc adr= C04F
pset function bit= 2
inbc adr= 4027  outbc adr= 4027  inbc adr= 8027  outbc adr= 8027  inbc adr= C027  outbc adr= C027

x=319 y=  0
point function bit= 2
inbc adr= 404F  inbc adr= 804F  inbc adr= C04F
pset function bit= 1
inbc adr= 4027  outbc adr= 4027  inbc adr= 8027  outbc adr= 8027  inbc adr= C027  outbc adr= C027

x=  0 y=  1
point function bit= 128
inbc adr= 5000  inbc adr= 9000  inbc adr= D000
pset function bit= 128
inbc adr= 4800  outbc adr= 4800  inbc adr= 8800  outbc adr= 8800  inbc adr= C800  outbc adr= C800

x=  1 y=  1
point function bit= 32
inbc adr= 5000  inbc adr= 9000  inbc adr= D000
pset function bit= 64
inbc adr= 4800  outbc adr= 4800  inbc adr= 8800  outbc adr= 8800  inbc adr= C800  outbc adr= C800
```

```
x= 2 y= 1
point function bit= 8
inbc adr= 5000  inbc adr= 9000  inbc adr= D000
pset function bit= 32
inbc adr= 4800  outbc adr= 4800  inbc adr= 8800  outbc adr= 8800  inbc adr= C800  outbc adr= C800

x= 3 y= 1
point function bit= 2
inbc adr= 5000  inbc adr= 9000  inbc adr= D000
pset function bit= 16
inbc adr= 4800  outbc adr= 4800  inbc adr= 8800  outbc adr= 8800  inbc adr= C800  outbc adr= C800
```

例では，デバッグにCDBを用いずスタブを利用し，リストの所々に変数などの値を表示させる行を加えています．

関数reduceがメインプログラムです．この関数は単純にyとxを制御変数として2重ループをしているだけです．この中で呼び出されているのがpsetとpoint関数です．BASICではこのステートメントをもっていますからこのまま実行する事ができますが，BDS Cにはありませんからこの部品もCで記述します．

psetとpoint関数は，reduce2.cのファイル中で定義されています．この2つは完全にハードに依存しますからシャープX1シリーズ以外では書き直す必要があります．このようにハードに密着した部分もCで記述できますが，一般にはBASICの内部ルーチンを利用するのが簡単です．

特にX1シリーズではグラフィク画面の座標とVRAMアドレスの対応が複雑ですので高級言語での記述はかなり長くなり，実行も遅くなります．実用的にはアセンブラで作成するか，BASICのルーチンをCALLしてください．

reduce2.cファイルの最後の3つの関数はデバッグ用です．mainは関数reduceへ導くだけの役割しか持っていません．この理由については本項の後の方で説明します．

次のinbc, outbcはI/O領域との入出力を行なう関数です．元々，BDS Cには入出力用にinp, outp関数が用意されていますが，シャープX1ではZ80を用いてI/Oを64Kバイトに拡張してあるため，ここでは使えません．あとでアセンブラ定義する事を前提にスタブだけ記述しておきます．

プログラムのコンパイルはいつもと全く同じです．ただ，例ではリンクにL2を使っています．別にCLINKを用いても良いのですが，main関数がサブファイルreduce2.cにあるため，

```
CLINK reduce reduce2
```

とすると，reduce.cファイルの中にmain関数がないというメッセージが出て処理が中断されます．本来はリンクの順序を逆にして出力ファイル名を指定するのが一番簡単な方法でしょう．

スタブによるデバッグは実行あるのみです．ここでは，ループカウンタx，y，それにinbc，outbcを実行した時にその引数（I／Oアドレス）を表示し，正しいかどうかのチェックを行ないます．2重ループですのでかなり時間がかかりますが，適当な所まで表示してみます．

正しい動作が確認されたら，ターゲットシステムで実行させてみましょう．

操作例7-9　reduceのターゲットへの変更

A>type reduce.h

〈reduce.h〉

```
int      x,y;     /* location      */
int      ty;
char     bitmask;
char     r,g,b;
unsigned          adr;

#define DEBUG  0
```

デバッグモードの解除．

A>type reduce3.csm

〈reduce3.csm〉

アセンブラ定義関数を作成する．長い関数はやはりCP/M上でデバッグしておく必要がある．

Z80アセンブラで定義してある．

```
;main()
;        This main function must be
;        called directly by BASIC.

         FUNCTION       MAIN
         EXTERNAL       REDUCE

         LD       (SPBF),SP      ;SAVE BASIC STACK
         LD       SP,0F800H      ;X1 FREE AREA
```

```
        CALL    REDUCE
        LD      SP,(SPBF)
        RET
SPBF:   DS      2

        ENDFUNC


;outbc(adr,byte)
;adr:  io address
;byte: out data

        FUNCTION        OUTBC

        LD      IX,2
        ADD     IX,SP
        PUSH    BC

        LD      C,(IX+0)        ;FIRST ARG
        LD      B,(IX+1)

        LD      A,(IX+2)        ;2ND ARG
        OUT     (C),A

        POP     BC
        RET

        ENDFUNC

;inbc(adr)
;adr:  io address

        FUNCTION        INBC

        POP     DE
        POP     HL
        PUSH    HL
        PUSH    DE
        PUSH    BC

        LD      C,L
        LD      B,H

        IN      L,(C)
        LD      H,0
```

```
        POP     BC
        RET

        ENDFUNC
```

```
A>cc reduce -me000 -ef06f
BD Software C Compiler v1.50a (part I)
  35K elbowroom
BD Software C Compiler v1.50 (part II)
  32K to spare

A>cc reduce2 -me000 -ef06f
BD Software C Compiler v1.50a (part I)
  35K elbowroom
BD Software C Compiler v1.50 (part II)
  31K to spare


A>submit zcmr reduce3 ←―――――――― reduce3.csm のアセンブル

A>ZCASM REDUCE3
ARMAT Z80-CRL preprocessor v1.0
Pass 1: End
Pass 2:
 -Processing the MAIN function..
 -Processing the OUTBC function..
 -Processing the INBC function..
  REDUCE3.ASZ is ready to be assembled.

A>MRASM REDUCE3.AAZ
Armat Z80 Assembler - VER 1.1
  (c) Armat co. 1985 ALL Rights Reserved.

No Fatal error(s)

A>LOAD REDUCE3

FIRST ADDRESS 0100
LAST  ADDRESS 0359
BYTES READ    006E
RECORDS WRITTEN 05
```

```
A>ERA REDUCE3.CRL
NO FILE
A>ERA REDUCE3.ASZ
A>REN REDUCE3.CRL=REDUCE3.COM◄──────reduce3.crl が完成.

A>l2 reduce reduce2 reduce3 -org e000 -t f800 -w◄──プログラムスタートアドレス E000H
                                                    スタック F800H
Mark of the Unicorn Linker ver. 2.2.2               シンボルファイル作成を指定してL2で
                                                    リンク.(CLINKでも可能)
Loading  REDUCE.CRL
Loading  REDUCE2.CRL
Loading  REDUCE3.CRL
Scanning DEFF.CRL
Can't open DEFF2.CRL◄──────DEFF2.CRL のライブラリは作成して
                           いないので無視される.
Link statistics:
  Number of functions: 7
  Code ends at: 0xEAF5
  Externals begin at: 0xF06F
  Externals end at: 0xF07B
  End of current TPA: 0xDC06
  Jump table bytes saved: 0x24
  Link space remaining: 25K

A>type reduce.sym

E7B8 REDUCE     E829 PSET     E998 DECODE     E9DD POINT
EAC3 MAIN       EAD4 OUTBC    EAE8 INBC

A>    └──────このアドレスが求めるエントリーアドレス.
```

操作例7-9がターゲットシステムへの移植手続きです．まず，CP/Mと異なる部分を変更し，新たにプログラムが必要な部分は作成します．

reduce.h ……デバッグモードを解除します．
reduce.c ……変更はありません．
reduce2.c ……変更はありません．
reduce3.csm……新たに作成します．

reduce2.cではアセンブラ定義関数をデバッグ用関数で置き換えてありましたが，これらは条件コンパイルの指定になっていますから，デバッグモード

の解除で自動的に削除されます．従って新たにreduce3. csmというファイルを作り，main, inbc, outbcそれぞれの関数をアセンブラで記述します．

この3つは，すべてZ80の機能を利用しなければならない関数なので，ここでは私の作成したZCASMを用い，Z80ザイログニーモニックで記述しました．ROM化でも，他OS用プログラムの作成でも現実にはZ80AなどのCPUを用いる場合の方が圧倒的に多いと思います．自画自讃ですが，Z80の機能をすべて使えるZCASMは非常に重宝します．

さて，ここで述べておかねばならない重要な事があります．それはmain関数を何故アセンブラで定義しているかという事です．それは，このreduceというプログラムが完結したものではなく，BASICという他のインタプリタ言語の1サブルーチンとして動作するようにしなければならないからです．

実は，この機能は本来のBDS Cにはありません．BDS Cで記述されたプログラムはあくまでBDS Cで記述されたプログラムからしか呼び出す事ができません．今回のプログラムがE000番地から始まっている事は指定したのですから当然ですが，BASICからE000番地をCALLしても期待したとおりに動作してくれないのです．

これは，BDS CではランタイムパッケージC. CCCの頭でスタックを新規に設定するからで，サブルーチンとして呼び出されて処理を終えても呼び先に正しく戻りません．この時スタック設定のアドレスを任意の番地に変更するリンクオプション”t”を用いても意味がありません．

なお，注意しなければならないのはCLINKのオプション”n”です．この指定はCP／M上でプログラムが実行終了した時にリブートさせないようにする事ができます．しかしCP／MのコマンドプロセッサーであるCCPがBDS CプログラムをCALLしたあと，そのスタックポインターをRAMに記憶しておいてスタックを再設定する方法を用いており，その再設定が6，7番地のアドレス（BDOSへのジャンプベクター）から計算して行なわれるため，ROM化などの用途には使えません．しかも，C. CCCをROM用に変更してしまうとこの機能そのものが失われます．nオプションが指定でき，かつスタックの

の再設定が任意のアドレスで行なえるようになっていると良いのですが……

今回はプログラムを頭から（E000番地）から実行せず，main関係をじかに呼び出す事でこの問題を解決する事にしました．つまり，

C.CCCの初期化ルーチン

main関数

の順で実行する所を，直接mainをCALLしてしまうのです．mainのエントリアドレスはリンク時にシンボルファイル出力を指定しておき，symファイルを見ると分ります．

ただしこの方法を用いるとBDS Cの機能が一部利用できなくなります．次のような点に気を付けてください．

(1) inp，outp関数は使えません．

BDS Cではinp，outp関数を実現するためRAM領域にプログラムを書き込みます．この操作は初期設定で行なっているためinp，outpは使えなくなります．

(2) 外部変数の初期化機能は利用できません．

(3) スタックの設定（オプション－t）は無効になります．

(4) プログラムのスタートアドレスがコンパイルのたびに変わります．

今回は次のようにmain関数を作成しました．

```
        LD      (SPBF),SP  スタックポインターの退職
        LD      SP,0F800H  スタックの新規設定
        CALL    REDUCE     reduce関数のCALL
        LD      SP,(SPBF)  スタックの回復
        RET
SPBF:DS         2          スタック退避用メモリ
```

ここでSPBFはスタックの退避用メモリバッファですが，RAMでなければなりませんから，本来は外部変数として登録しておき参照する形をとらなければなりません．しかしROM化ではありませんからそこまで厳密には行ないませんでした（ROM化ならこのような操作そのものが必要ない筈です）．

outbc関数はIXレジスターを用いて引数の参照をおこなっています．inbcは引数が一つしかありませんからPUSH，POPで参照しています．操作例7-9を参照してください．

さて，コンパイル，リンクが終われば，ターゲットシステムで実行します．実際の開発作業ではターゲットシステムにプログラムを移す操作そのものが問題になります．これはその都度方法を考える必要がありますので具体的に触れる事ができませんが，通常は開発を始める前にその方法を検討しておきます（場合によっては，そのためにプログラムを何本も作る必要があります）．

今回は，BASICでCP／Mディスクのファイルを読むプログラムを作成し，プログラムを転送しました．

さて，実行ですがアセンブラ定義部分が短く，殆どデバッグする必要はありませんでしたが，一般には転送あるいはROM焼きの作業が繰り返されます．ターゲット上でしかデバッグできない部分は極力減らし，CP／M上で多くの部分をデバッグしておくのがこつと言えそうです．

さて，結果は大変残念なものとなりました．BASICでは遅い処理を高速化するのが目的だったのですが，かえってBASICより遅くなってしまいました．これはBASICそのものはアセンブラで作成されていますから，pset，point命令の処理がはるかに高速なためだと思われます．

〈実行時間〉

BASIC　3分35秒（215秒）

BDS C　7分2秒（422秒）

比較したBASICプログラムをリスト7-10に示します（X1ターボBASIC, CZ8FB02).

リスト7-10 reduceのBASICプログラム

```
10 DEFINT A-Z
20 FOR Y=0 TO 99
30 FOR X=0 TO 319
40 PSET(X,Y,POINT(X*2,Y*2))
50 NEXT
```

7-4 インサーキットエミュレータを用いる場合

最近はプログラムの開発にインサーキットエミュレータ(ICE）やROMシミュレータを使い，効率を上げる事も多くなりました．かつてはMDS（Microcomputer Development Systemインテルの開発マシン．CP／Mはもともとこの MDSを最初のターゲットとして開発されました）などの高価なマイコン開発ツールでなければICEを用いる事ができず，ちょっとした開発作業には利用できなかったものですが，最近はCP／Mマシンをホストにした8085やZ80用の安価なものが市販されており，個人でも利用できるようになりました．

開発環境は抜群に良くなった訳ですが，安価なICEでは開発ソフトはまだ充実しているという所まではいっていません．アセンブラについてはともかく，高級言語についてはまだ皆無という状況です．従って，ICEを用いてBDS Cのプログラムを開発するのはなかなか難しいと言わざるを得ません．CDBなどのような専用のデバッガーがICE用にあればベストなのですが，一般にはアセンブラ用のシンボリックデバッガーを用いる事になります．

CP／M上でSID，ZSIDを使う場合も同様ですが，シンボリックデバッガーを用いた場合次のような点に注意します．

(1) リンクはL2で行なう

これは，CLINKを用いると他の関数の呼出しが関数の先頭にあるダミージャンプテーブルを介して行なわれるため，逆アセンブルした時にCALL命令がどの関数にジャンプするのか分りにくくなります．L2では直接関数をCALLするように変更されます．

(2) コンパイル時に“－e”オプションをつける

オプションを付けると外部変数の参照の際，じかに変数アドレスが命令中に展開されます．逆アセンブルで処理内容が把握しやすくなります．

(3) コンパイル時に“－o”オプションをつける

これも(2)と同様な理由からです．引数およびローカル変数の参照でSDLI, LDLIなど直接数値をオブジェクトに展開する処理がおこなわれなくなります．

以上の3点に気を付ける事で，逆アセンブルが読みやすくなりますので，プログラムのトレースがたやすくなります．勿論これらはデバッグ時にのみ必要な事ですから，最終的なオブジェクトとは関係ありません．

さて，この方法では関数名はシンボルでおこなえるのですが，変数名は参照できませんからかなり不便です．やはり前項のようにCP／M上でCDBなどを用いて関数単位でのデバッグを進めておき，アセンブラ定義部分を主体にICEなどのデバッガーで開発をするのが得策だと思います．

なお，ICEやPROMライターによってはCOMファイルを受け付けず，HEXファイルでなければ使えないものがあります．単純にターゲットシステムへプログラム転送をする場合でも一番簡単なのはRS-232Cポートを用いて，HEXファイルを送る方法です．BDS Cリンカーには HEXファイルを出力する機能はありませんから，COMファイルをHEXファイルに変換するユーティリティプログラムがあると便利です．

この機能を持ったプログラム（GENHEX）を作成しておきましたので，必要な方は利用してください．

リスト 7-11　genhex.c

リストを入力する時は，この行番号は入力しない．

```
 1: /*****************************************************************
 2:
 3:                 -----   GENHEX.C  -----
 4:
 5:         HEX file generator from COM file
 6:
 7:         Author: Tsu.Mitarai     28/dec/1985
 8:
 9:                 A>cc genhex -e3000 -o
10:                 A>clink genhex
11:                         (or A>l2 genhex)
12:
13: *****************************************************************/
14:
15: #include        <bdscio.h>
16:
17: #define TITLE   "HEX file generator (c) Armat co. 1985¥n"
18: #define ERROR   -1
19: #define INEXT   ".COM"
20: #define OUTEXT  ".HEX"
21: #define CPMEOF  0x1a
22: #define BUFSIZE 128             /* input file buffer size (Don't change) */
23: #define LINSIZE 16              /* bytes par line       */
24:
25: /***    output file     ***/
26:
27: char    outname[20];
28: FILE    fpout;                  /* output file buffer   */
29:
30: /***    input file      ***/
31:
32: char    inname[20];
33: char    fin[BUFSIZE];           /* input file buffer    */
34: int     fd;
35:
36: /***    other externals ***/
37:
38: unsigned hexadr;                /* hex file output address      */
39:
```

```
40:
41: main(argc,argv)
42: int     argc;
43: char    *argv[];
44: {
45:         printf (TITLE);
46:
47:         if (argc != 2 && argc != 3)
48:                 kexit("Usage: GENHEX filename [address]¥n");
49:
50:         strcpy (inname, argv[1]);
51:         strcat (inname, INEXT);                         /* input file' */
52:
53:         strcpy (outname,argv[1]);
54:         strcat (outname,OUTEXT);                        /* output file */
55:
56:         hexadr = 0x100;                                 /* TPA address */
57:         if (argc == 3)  sscanf (argv[2], "%x", &hexadr);
58:
59:         if (ERROR == (fd = open (inname, 0)))           /* input file open */
60:                 kexit( "File not exist" );
61:
62:         if ( ERROR == fcreat (outname, fpout))          /* output file create */
63:                 kexit ( "Directory full?" );
64:
65:         genhex();                                       /* this is main routine */
66:
67:         close (fd);
68:         fclose (fpout);
69:         printf ( "Genhex complete" );
70: }
71:
72: /***    Message output, break submit & exit     ***/
73:
74: kexit(pnt)
75: char    *pnt;
76: {
77:         printf ( pnt );
78:         unlink ("$$$.SUB");
79:         exit();
80: }
81:
82: /***    Main routine    ***/
83:
84: genhex()
```

```
 85: {
 86:         char    *cp;
 87:         while (read ( fd, fin, BUFSIZE/128 ) >0 )
 88:         {
 89:                 for ( cp=fin; cp < fin+BUFSIZE; cp+=LINSIZE )
 90:                 {
 91:                         hexline ( cp, hexadr );
 92:                         hexadr += LINSIZE;
 93:                 }
 94:         }
 95:         if (ERROR == fprintf ( fpout, ":00000001FF¥n%c",CPMEOF ))
 96:                 Kexit ( "Disk full" );
 97: }
 98:
 99: /***    1 line output routine   ***/
100:
101: hexline( buf, adr )
102: char    *buf;
103: unsigned        adr;
104: {
105:         int     i;
106:         int     cksum;
107:
108:         if (ERROR == fprintf (fpout, ":%02x%04x00", LINSIZE, adr))
109:                 kexit ("Disk full");
110:         cksum = (adr/256) + (adr&255) + LINSIZE;
111:         for (i=0; i<LINSIZE; i++)
112:         {
113:                 cksum = cksum + *buf;
114:                 if (ERROR == fprintf (fpout, "%02x", *buf++))
115:                         Kexit ("Disk full");
116:         }
117:         if (ERROR == fprintf (fpout, "%02x¥n", (-cksum)&255))
118:                 kexit ("Disk full");
119: }
```

<GENHEXの使い方>

A>GENHEX　ファイルネーム　[アドレス]

ファイルネームはCOMファイルに限ります．（".COM"は記述しません）アドレスを指定する場合は16進数4桁以内とし，そのアドレスからHEXファイルが始まります．アドレスを省略した時はデフォルト値100Hからとなりま

す．

操作例 7-12　genhex の使い方

```
A>cc genhex -e3000 -o
BD Software C Compiler v1.50a (part I)
  33K elbowroom
BD Software C Compiler v1.50 (part II)
  30K to spare

A>clink genhex
BD Software C Linker   v1.50
Linkage complete
  39K left over

A>genhex genhex ←───────────── genhex.com 自身を HEX ファイルに
HEX file generator (c) Armat co. 1985      変換してみる．
Genhex complete

A>type genhex.hex

:100100002A0600F90000C30C01C30000CD6203CD34 ┐
:10011000B808C37B040030B807EE1EB934C37704B7 │
                    ¦                        │ HEX ファイルは
                    ¦ （中略）               │ 正しく作成されている．
                    ¦                        │
:101EE000D2E81EE3732372EB01CD3501C1C91A1AB2 │
:101EF0001A1A1A1A1A1A1A1A1A1A1A1A1A1A1AE379 ┘
:00000001FF

A>load genhex ←────────────── HEX ファイルを COM ファイルに
                                 変換する．
FIRST ADDRESS 0100
LAST  ADDRESS 1EFF
BYTES READ    1E00
RECORDS WRITTEN 3C

A>genhex genhex ←───────────── 変換後の genhex.com で
HEX file generator (c) Armat co. 1985      もう一度 HEX ファイルに変換してみる．
Genhex complete ←───────────── 正しく動作した．
A>
```

7-5 インタラプト処理の方法

割り込み（インタラプト）はマイコンシステムにおいて多用される基本的な手法の一つであり，現在では時間の計測，通信，ディスク制御などで割り込みを使わないシステムは考えられませんが，そのプログラム作成はかなり難しいものです．私の場合同じ大きさのプログラムなら，内容がある程度以上高度であれば，割り込みを使ったものは使わないものに比べ，数倍の開発期間を設定します．

「高度」というのはかなり漠然とした表現ですが，一つのI／Oデバイス（例えばディスク）をメインプログラムでアクセスしている最中に，割り込み側からも同じI／Oデバイスを読み書きするプログラムを作成する作業を想像していただければその困難さが理解できるでしょう（マルチタスクになるわけです）．

このようなプログラムはなるべく避けるのが得策ですが，現実にはなかなかそうもいきません．そのような時，プログラム容量と実行速度に余裕があれば，BDS Cを使うと若干ですが開発しやすくなるでしょう．

割り込みを用いていても，メインプログラム，割り込み側プログラム共その作成は通常の場合とかわりありません．ただし，割り込む際にレジスタの値を退避し，割り込みプログラムを呼び出す必要がありますから，そこだけはアセンブラで作成する必要があります．

アセンブラルーチン（interupt. csm）のサンプルをリスト7-13に示します．

リスト 7-13 interupt. csm

```
FUNCTION        INTERUPT
EXTERNAL        INTMAIN

PUSH    PSW
PUSH    D
PUSH    H
CALL    INTMAIN
POP     H
POP     D
```

```
        POP     PSW
        EI
        RET

        ENDFUNC
```

この例では，A，DE，HLレジスタとフラグの内容を退避し，割り込みルーチンである関数intmainを呼び出しています．このプログラム自身はinteruptという関数になっています．Cだけで割り込みルーチンを書く限りBCレジスタの値は変化しませんので退避する必要はありません．Z80なら，裏レジスタを使っても良いでしょう．勿論アセンブラで作成したZ80関数などで他のレジスタを使っていれば，そのレジスタも退避しなければなりません．関数の最後でEIを実行し，インタラプトを許可しておきます．

次に，メインプログラム，インタラプトのメインプログラム，及びinterupt.csmをコンパイルし，リンクします．このとき，完成したプログラムの中には，interupt関数を呼び出すルーチンがないので，割り込み側プログラムは実行される事がありません．そこで，これを呼び出す部分を作成します．38H番地が割り込みのかかるアドレスとすると，38H番地にinterupt関数へのジャンプ命令を書き込みます．この方法には次のようなものがあります．

(1)　メインプログラムの最初に，ジャンプ命令そのものを38H番地に書き込むルーチンを記述しておく方法

最も簡単で，手作業に頼る必要もありませんが，38-3AH番地はRAMでなければなりません．

(2)　DDTなどのデバッガーを利用してジャンプ命令を書き込む方法

DDTでプログラムをオフセット付きでロードし，38H番地にあたる場所にジャンプ命令を書き込みます（リスト7-14参照）．

リスト7-14 インタラプト時のジャンプ命令の書き込み

（注：プログラム本体は100Hからとする）

```
A>ddt
DDT VERS 2.2

-f100,fff,0 ←―――― あらかじめ利用するエリアをクリアしておく.

-iinttest.com ←――┐ テストプログラム（100Hから動作する）を100番地オフセットをつけて
-r100 ←―――――――――┘ 200Hからロードする.
NEXT  PC
0A00 0100

-a100
0100  jmp 100 ←―――― 100H番地（本来の0番地）に100番地への
0103                ジャンプ命令を書き込む.

-a138
0138  jmp 3b2 ←―――― 138H番地（本来の38H番地）に
013B                関数interuptへのジャンプ命令を書き込む.
-^C

A>save 10 inttest.com ←―――― プログラム全体をセーブする.
A>
```

以上のような方法が考えられますが，(1)はRAMでなければならないので使えない場合があります．また(2)は手作業を必要とします．特にinterupt関数のエントリ番地がコンパイル，リンクのたびに変化すると，サブミットファイルにする事ができませんから，どうしても誤りが発生しやすくなります．

そこで，L2リンカーを使用し，必ずinterupt関数が最初にリンクされるようにするとinteruptのエントリアドレスが必ずランタイムパッケージC.CCCの次のアドレスとなって変化しなくなりますから一連の操作をサブミットファイル化する事ができるようになります．CLINKではコマンドラインで指定したとおりにリンクせず，main関数を先頭に持ってきますから，この場合には不都合です．

インタラプトが2つ以上かかる場合にも割り込み呼出し用の関数だけを頭に並べておいてL2を使えば，アドレスは固定となります（勿論，これらの関数を修正した時は変化します）．

インタラプトはBDS Cではサポートしていないために前記のように面倒な手続きを行なわなければなりませんが，手順さえ飲み込んでしまえばプログラム自体の作成は簡単になりますから，是非試してみてください．

プログラムのデバッグについては，試行錯誤の部分が多くなります．CP／M上で行なう関数テストがどれだけ役に立つか疑問です．特にインタラプトの許可，不許可（EI，DI）は関数で実現する方法が良く用いられますが，効率が悪いうえプログラムのトレースは殆どできません．

ICEがあればかなり楽になると思います．しかし，メインプログラム→インタラプトのデータ受渡しに外部変数を使った場合，BDS Cでは一般に2バイトを別々に書き込むので1バイトだけ書き込んだ段階で割り込みがかかってしまうと，全くデタラメなデータを受け取ってしまうことになります．

－eオプションを使えば，1命令で2バイト読み書きするようになりますので少しは楽になりますが，プログラムによって1バイトずつのアクセスになる場合がどうしても発生します．

この段階までくれば，BDS Cでは最早お手上げです．他の開発法を検討するべきでしょう．

あとがき

C言語の流行ぶりには目を見張るものがあります．本来C言語はUNIXの開発に使われたものですが，すでにUNIXを離れて一人立ちしてしまいました．マイクロコンピュータの世界でもかつてはアセンブラでしか作成されていなかったOSや制御プログラムもどんどんCに置き換えられ，その威力を遺憾なく発揮しています．

C言語がこれだけ歓迎されたのは，得られるオブジェクトが高速で小さく，さらに最新のプログラム理論を取り入れながらも，規則に縛られる事なくかなりラフな記述もできてしまう融通性を持っていたからだと思います．小文字表記ですから見た目にも美しく，演算記号を使う事でタイプの量もかなり減ります．覚えるまで少し時間がかかるという欠点はありますが，一度覚えると実に便利な道具となります．

Cは様々なCPU，OS用に開発されており，あらゆるマシン上で走行しています．恐らく，今後Cは標準的な言語となり，アセンブラにとって替る分野もさらに多くなるでしょう（そのためにはもう少し環境が整備される必要があります）．

C言語を学ぶ方にとって，BDS Cおよびα-Cは格好の素材です．是非，本書とBDS Cあるいはα-CでC言語をマスターされる事を期待しています．

なお，本書はBDS Cのマニュアルではありません．あくまで，マニュアルでは足りない部分を補う参考書である事をお断りしておきます．

また，BDS Cの解析などについては一部情報を参考にした部分もありますが，その殆んどは独自に解析したものであり，その内容についてはすべて著者に責任があります．

最後に，本書執筆の機会を与えていただいた工学図書株式会社の方々，ソフトウェアと情報を提供していただいた株式会社ライフボートの方々に厚くお礼申し上げると共に，執筆に協力してくれた妻，理英に感謝します．

御手洗　毅

付　録

■標準関数リスト■

1.　一般関数

関数名	機能の概略	参照ページ
exit	プログラムの終了	52
bdos	BDOS コールを行なう	52
bios	BIOS コールを行なう	52
biosh	BIOS コールを行なう	52
peek	メモリの内容を読み出す	52
poke	メモリに書き込む	53
inp	I/O ポートから入力する	53
outp	I/O ポートに出力する	53
pause	キーが押されるまで待つ	53
sleep	指定時間，処理を中断する	53
call	アセンブラサブルーチンの CALL	53
calla	アセンブラサブルーチンの CALL	54
abs	絶対値をとる	54
max	2つの数の大きい値を取る	54
srand	乱数の初期値を設定する	54
srandl	乱数の初期値を設定する	54
rand	乱数を発生する	54
setmem	メモリを定数で充たす	55
movmem	メモリのブロック移動をする	55

関数名	機能の概略	参照ページ
qsort	ソートを行なう	55
exec	プログラムをロードし，実行する	57
execl	プログラムをロードし，実行する	58
execv	プログラムをロードし，実行する	58
swapin	ファイルをロードする	58
codend	プログラムの最終アドレスを得る	58
externs	外部変数の先頭アドレスを得る	58
endext	外部変数の最終アドレスを得る	59
topofmem	フリーエリアの先頭アドレスを得る	59
alloc	フリーメモリを得る	59
free	メモリを解放する	62
sbrk	フリーメモリを得る	62
rsvstk	フリーメモリとスタックの距離を設定する	62
setjmp	longjmp を初期化する	62
longjmp	関数外へジャンプする	62
memcmp	メモリをブロック単位で比較する	65

2. 文字入出力関数

関数名	機能の概略	参照ページ
getchar	標準入力から1文字入力	66
ungetch	標準入力へ1文字もどす	66
kbhit	キーボードのセンス	66
putchar	標準出力へ1文字出力	66
putch	コンソールへ1文字出力	67
puts	コンソールへ文字列出力	67

関数名	機能の概略	参照ページ
getline	コンソールから文字列入力	67
gets	コンソールから文字列入力	67
printf	フォーマット付き文字列出力	68
__spr	printf関数の本体部分	69
lprintf	リスト装置に対するprintf	72
scanf	フォーマット付き文字列入力	72

3. 文字列処理関数

関数名	機能の概略	参照ページ
isalpha	アルファベットか否かのテスト	72
isupper	大文字かどうかのテスト	73
islower	小文字かどうかのテスト	73
isdigit	数字かどうかのテスト	73
toupper	大文字への変換	73
tolower	小文字への変換	73
isspace	空白文字かどうかのテスト	73
sprintf	メモリに対するprintf	73
sscanf	メモリからのscanf	74
strcat	文字列の連結	74
strcmp	文字列の比較	74
strcpy	文字列のコピー	74
strlen	文字列の長さを求める	75
atoi	アスキー文字を数値に変換する	75
initw	int変数を初期化する	75
initb	char変数を初期化する	75

関数名	機能の概略	参照ページ
index	文字列のサーチ	76

4. 低レベルファイルI/O関数

関数名	機能の概略	参照ページ
open	ファイルのオープン	76
create	ファイルの作成	76
close	ファイルのクローズ	77
read	ファイルの読み込み	77
write	ファイルの書き込み	77
seek	ファイルr/wポインタの変更	77
tell	r/wポインタを求める	78
unlink	ファイルを消去する	78
rename	ファイル名を変更する	78
fabort	ファイル読み込みを中断する	78
cfsize	ファイルサイズを求める	78
errno	ディスクエラーの種類を知る	79
errmsg	errnoに対応するメッセージ	79
setfcb	FCBにファイルネームをセットする	79
fcbaddr	FCBの先頭アドレスを求める	79

5. バッファードファイルI/O関数

関数名	機能の概略	参照ページ
fopen	ファイルをオープンする	80
getc	ファイルからの1文字入力	80

関数名	機能の概略	参照ページ
ungetc	入力ファイルに1文字戻す	80
getw	ファイルからのワード入力	80
fcreat	ファイルの作成	81
putc	ファイルへの1文字出力	81
putw	ファイルへのワード出力	81
fflush	ファイルバッファの書き込み	81
fclose	ファイルのクローズ	81
fprintf	ファイルへのprintf	82
fscanf	ファイルからのscanf	82
fgets	ファイルからのgets	82
fputs	ファイルへのputs	82
fappend	ファイルの連結	82

■浮動小数点パッケージの関数■

関数名	機能の概略	参照ページ
fpadd	浮動小数点数の加算	95
fpsub	浮動小数点数の減算	95
fpmult	浮動小数点数の乗算	95
fpdiv	浮動小数点数の除算	95
atof	文字列の浮動小数点数への変換	95
ftoa	浮動小数点数の文字列への変換	95
itof	int変数の浮動小数点数への変換	96
fpcomp	浮動小数点数同士の大小比較	96
printf	浮動小数点数用のprintf	96

■倍精度整数パッケージの関数■

関数名	機能の概略	参照ページ
ladd	倍精度整数の加算	99
lsub	倍精度整数の減算	99
lmul	倍精度整数の乗算	99
ldiv	倍精度整数の除算	99
lmod	倍精度整数の剰余を求める	99
atol	文字列を倍精度整数に変換する	100
ltoa	倍精度整数を文字列に変換する	100
itol	整数を倍精度整数に変換する	100
ltoi	倍精度整数を整数に変換する	100
utol	符号なし整数を倍精度整数に変換する	100
ltou	倍精度整数を符号なし整数に変換する	100
lcomp	倍精度整数同士の大小比較	100
lassign	倍精度整数の代入	101

■printfの変換文字■

変換文字	機能の概略
%d	10進整数に変換する
%u	符号なし10進整数に変換する
%c	アスキー文字を表示する
%s	文字列の出力
%o	8進数で出力する
%x	16進数で出力する

■文字列中の特殊文字■

文　字	機能の概略
¥n	ラインフィード（LF）
¥t	タブ
¥b	バックスペース
¥r	キャリジリターン（CR）
¥f	フォームフィード
¥¥	¥記号
¥'	クォーテーションマーク
¥"	ダブルクォーテーションマーク
¥0	NULL（0）
¥ddd	アスキーコードの指定（8進数）

参　考　文　献

1.　プログラミング言語C：B.W.カーニハン・D.M.リッチー共著
（共立出版）　　　　　石田晴久訳

2.　BDS C Compiler User's Guide：Leor Zolman著
（ライフボート）

3.　BDS Cユーザーズマニュアル：リオー・ゾールマン著,
（ライフボート）　　　川辺一生訳

4.　BDS Cの使い方　　　：稲垣幸則・渡辺　修共訳
（工学図書）

5.　C言語入門　　　　　：レス・ハンコック・モーリス・クリーガー共著
（アスキー）　　　　　アスキー出版局監訳

6.　インターフェース　1982年7月号
「システム記述言語Cとその応用－Cシステム比較試用記」：前田英明
（CQ出版社）

7.　インターフェース　1983年7月号
「ROM化可能なコンパイラ徹底研究－C（BDS C）」：久保雅彦
（CQ出版社）

8.　インターフェース　1984年1月～9月号
「マイコンソフトウェア作法の研究」：祐安重夫
（CQ出版社）

9. マイコンピュータ　1983年10月号
「入門特集C言語の研究」:
（CQ出版社）

10. プロセッサ　1985年5月～1986年4月
「C＋アセンブラのプログラミングテクニック」:小池慎一
（技術評論社）

索　引

≪和　文≫

≪欧 文≫

【A】

【B】

【C】

【D】

【E】

【F】

【G】

ディスクサービスについて

本書で紹介したプログラムはリストをタイプして入力する事もできますが，入力を誤る場合も多く，余り良い方法とは言えません．そこで，読者の方々に本書のプログラムを収録したフロッピーディスクをお分けいたします．

御希望の方は現金書留か郵便小為替で，10,000円同封し，「BDS Cプログラミングディスク希望」と明記し，住所，氏名，年齢，電話番号，勤務先（学校名），使用システム，ディスクタイプを書いて，下記に御申し込みください．

ディスク内容

fplong.h	float，long 関数のヘッダーファイル
zcasm.c	casmのZ80版ソースプログラム
zcasm.com	casmのZ80版実行ファイル
zcasm.sub	zcasmをMACRO-80で利用する場合のサブミットファイル
zcmr.sub	zcasmをMR-ASMで利用する場合のサブミットファイル
mrasm.com	Z80ニーモニックのアブソリュートアセンブラ
genhex.c	comファイル→hexファイル変換ユーティリティプログラム
genhex.com	genhexの実行ファイル
cnm.c	CDB用の行番号発生プログラム
dacrl.com	CRLファイルをZ80アセンブラのソースファイルに変換するユーティリティ（コラム2参照）

Armat社製のZ80アセンブラMR-ASMが附属していますので，MACRO-80を持っていない方でもzcasmを利用できます．

価　格

10,000円（送料含む）

対象システム

5インチ2Dは殆どの機種で問題ありません．他のディスクタイプと，特殊なシステムをお使いの方はあらかじめお問い合せください．時間がかかる場合があります．

申し込み先

〒227 横浜市緑区荏田町473-5
㈲アーマット　ディスクサービス係
TEL 045-911-7427

■著者略歴

昭和30年 9 月24日生れ.
昭和53年慶應義塾大学工学部計測工学科卒業.
昭和53年カシオ計算機㈱入社後電子楽器開発に従事.
昭和58年㈱フジテレビジョン入社.
昭和61年㈲アーマット　開発部.

著書

「スーパーインポーズCP/M」 工学図書

著者の承
認により
検印省略

BDS Cプログラミング

昭和61年 6 月15日　初　版
昭和61年11月 5 日　第 3 版

著　者　御手洗　毅
発行者　笠原　洪平
発行所　工学図書株式会社
（営業所）東京都千代田区九段北1-14-15
〒102　電話　東京（262）3772番
振替　東京 7-13465番
印刷所　株式会社サンヨー

 ISBN4-7692-0154-0 C3058　**定価 2,600円**